Universidad Carlos III de Madrid
Escuela Politécnica Superior

Departamento de Ingeniería Informática

TRABAJO DE FIN DE GRADO:

AGENTES AUTOMÁTICOS PARA VIDEOJUEGOS CON MÁQUINAS DE ESTADOS Y AUTÓMATAS FINITOS

Grado en Ingeniería Informática

Autor: Álvaro Rubio Castellano
Tutor: Germán Gutiérrez Sánchez

Septiembre 2017

Hazlo o no lo hagas, pero no lo intentes.
- Maestro Yoda.

Agradecimientos

Han sido 5 años a cada cuál más complicado tanto en el ámbito académico como en el personal, muchas subidas y bajadas, pero siempre he tenido alguna forma de recuperarme.

Me gustaría dar las gracias, en primer lugar, a mis padres por permitirme estudiar esta carrera y que, pese a las distancias que hay en ocasiones entre nosotros, siempre han estado para apoyarme en lo que fuese.

A mi hermana, que ha sido alguien de quién he tenido mucho que aprender y que ha sabido entenderme en la mayoría de las ocasiones.

A mi primo Mario, que lleva acompañándome desde primero de la ESO hasta el final de esta etapa y que siempre ha sido un apoyo importante tanto dentro de la universidad como fuera.

A los amigos del instituto por enseñarme tanto. Y a los de toda la vida con los que he vivido muy buenos momentos que me han dado fuerzas para seguir adelante.

A mis compañeros de universidad, y también amigos, Guille y Álvaro por las risas y las comidas de ”empresa” y por conseguir hacer más llevadera la carrera. También Alberto, Majadas, Victor, Alejandro y Sofía que han conseguido hacer que aprenda mucho en todos los aspectos en estos últimos 2 años. Y a todos mis compañeros de carrera en general.

A los profesores que me han guiado hasta el punto en el que me encuentro ahora y a mi tutor por ayudarme en la elección del TFG y guiarme en la realización del mismo.

Y, por supuesto, a Honduco, un compañero, un amigo y mucho más. Alguien que conocí en primero, pero que no ha sido hasta tercero cuando empezamos a entablar una relación de amistad más profunda. Alguien que ha sido un gran sustento y apoyo, sobretodo este último año.

Gracias a toda la gente que ha pasado por mi vida, porque de todo y de todos algo se aprende.

Abstract

Since the term Artificial Intelligence (AI) has appeared for the first time in 1956, it has been defined in a numerous and varied way, without reaching a unique definition. Before defining this term, other scientists and thinkers had already considered the abstraction of the logical reasoning of the human being to apply it to machines, as was the case of Alan Turing and his Universal Machine, known as the Turing Machine.

It was in 1936 when the British mathematician defined and described the operation of the Turing Machine (TM). Based on the concept of an effective algorithm or process, Turing designed his machine as a device capable of performing an algorithm, supporting Gödel's theories by demonstrating that there are problems that can not be solved by a TM[1]. Thanks to the contributions made by the English scientist, the Turing Test was born, which has been used for years to determine the ïntelligence.ºf a machine.

As discussed above, an attempt has been made to give a concrete definition to artificial intelligence. Next, some of the definitions given to the AI can be found, depending on the approximation that is made of it[2]:

- Systems that think like humans

  "The exciting new effort to make computers think... machines with minds, in the full and literal sense" (Haugeland, 1985)

- Systems that act as humans

  "The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990)

- Systems that think rationally

  "The study of mental faculties through the use of computational models" (Charniak and McDermott, 1985)

- Systems that act rationally

  "The branch of computer science that is concerned with the automation of intelligent behavior" (Luger and Stubblefield, 1993)

With the creation of the first video games, the developers have looked for ways to get them as close to reality as possible. One of the main details that are required when making a video game is to make the intelligence of the characters that compose it is as similar as possible to that of a human, that is, to bring the behaviour of the game closer to the behaviour of nature. The AI is responsible for achieving this feat and it could be said that in recent years the progression of this has been quite high in this field.

Apart from video games, artificial intelligence is also applied in areas of everyday life, such as robots (ASIMO, NAO), autonomous cars (Tesla) or personal assistants such as Siri.

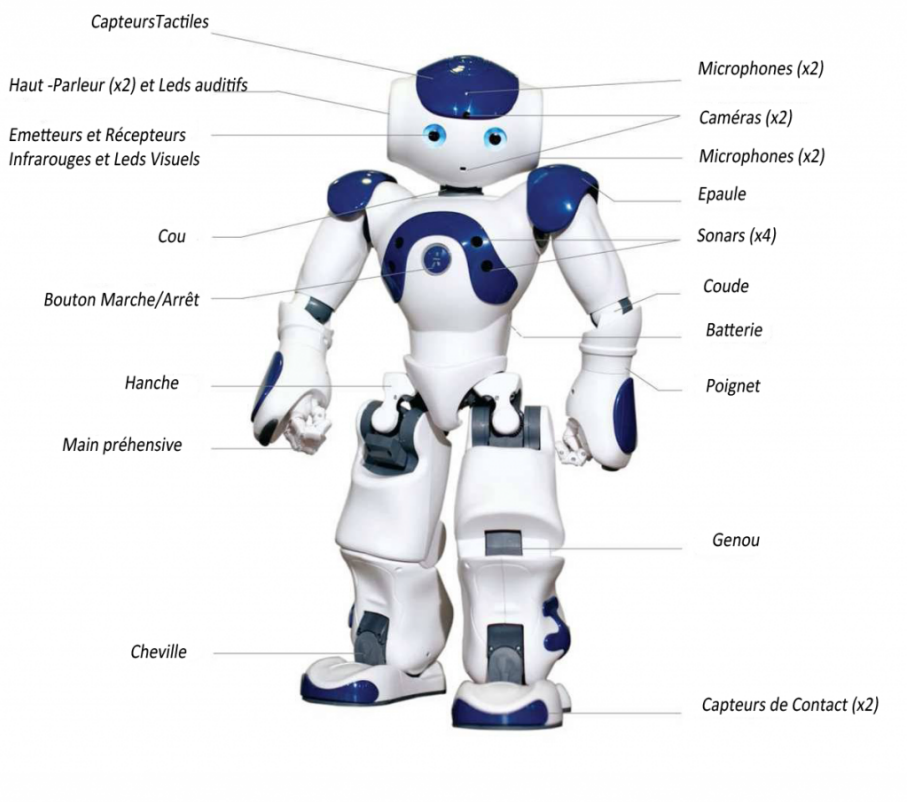

Figura 1: NAO robot with sensors (Source: `http://www.andorobots.com`).

The main objective of this project is to develop an automatic agent capable of controlling a system (character, machine, car, etc.) of a video game using finite state machines as technology. It is not about creating an application for later use in an open way, but about developing and studying an idea, observing how state machines can be applied in this field and comparing it to other similar projects. It is also considered, if possible, the participation of the automatic agent that can be developed in some type of competition.

The project is strictly governed by the license of the game that is used for the realization of the same, which in this case is the TORCS. This game is registered under the GNU General Public License[3] (GPL) in its most current version (3.0).

The work encompasses many topics of which it is important to know its current state to develop the project and to make a comparative. Among these topics are: automatic agents, finite state machines, video games, simulators and the game to be used as a platform in question.

A finite state machine, also known as a finite automaton, is an entity that is composed of one or more states and one or more transitions. The states represent the situation in which the system is controlled, while the transitions are the reflection of an action that has affected the system. In some cases, the transition may generate an output that may affect the operation of the machine in subsequent actions. The types of finite automaton that exist are two: deterministic (DFA) and non-deterministic (NDFA). The DFAS are characterized by having a single transition from one state to another with identical input; while in the NDFAs, with the same input, the state machine can move to 1 or more states from the same state. We can see examples of DFA and NDFA in the figures 2 and 3.

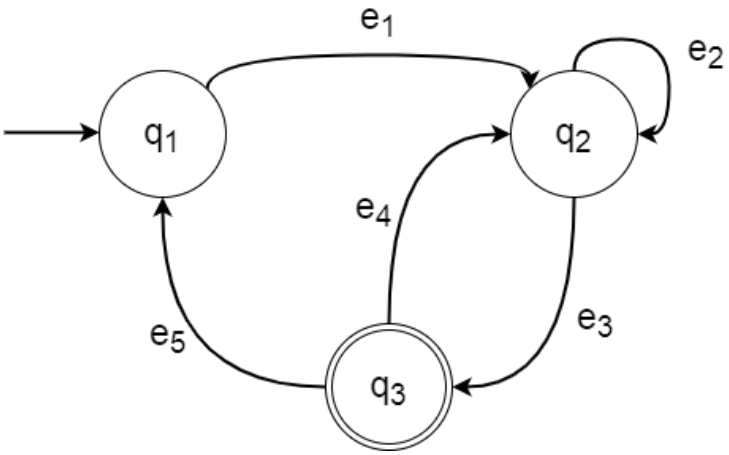

Figura 2: DFA example (Source: own).

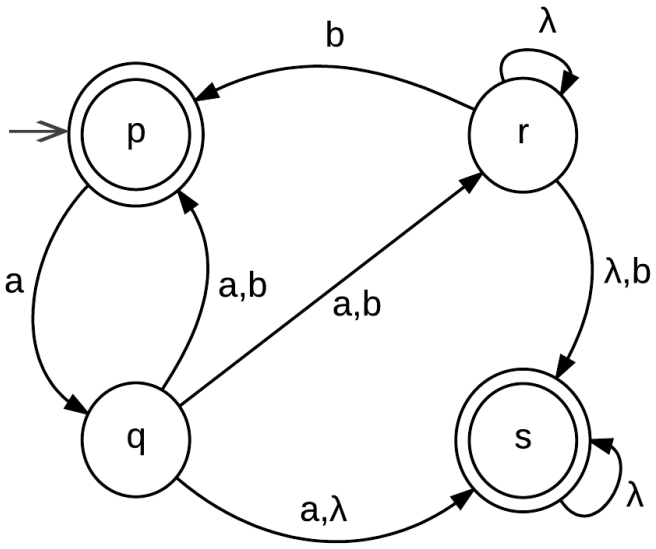

Figura 3: NDFA example (Source: own).

Given a NDFA it is always possible to find a DFA equivalent, that is, to recognize the same language. It should be noted that both types of finite automata are capable of solving the same type of problems. Other types of state machines that can be found, some of them with greater capacity of computation, are:

- Probabilistic automaton or Hidden Markov Model
- Pile automaton
- Turing Machine
- Cellular automaton
- Neural network

State machines have a large number of applications for both real life and simulated environments. Among these applications, it is important to highlight the following:

- **Electrical and electromechanical systems:** traffic lights, lifts, home appliances, vending machines, etc.
- **Recognition of natural language:** speech recognition, automatic translation, speech synthesis, etc.
- **Simulated environments:** video games and simulators.

During this project, the video game and simulator The Open Racing Car Simulator (TORCS) will be used. TORCS was created by Eric Espiè and Christophe Guionneau in 2004 and has been updated and improved thanks to the contributors (open source game) among which stands out to Bernhard Wymann. It is used to develop artificial intelligence in the motoring sector and a clear case of it are the competitions that are realized each year.

The design of the system is one of the most important parts of the work since it is the basis of the whole project. Therefore, a previous study of the language to be used in the agent programming has been done. Among the raised languages are: C, Matlab, Python and Java. Due to the preconditions of the project, the chosen language has been Java. On the other hand, the design includes the finite state machine model to be implemented. Since this is not a trivial problem, it has been necessary to make different designs and choose one of them.

In order to understand these designs, it is essential to know how the client's part is designed. This part has been obtained from a client that had already been programmed by a third person and has been adapted for the system to be developed. The client is part of an architecture that, following a patch that needs to be introduced into the game data, follows the client-server model. The server is connected to the game and allows direct communication with it, while the client is included in the agent that is developed and allows to obtain the state of the game through sensors. The type of protocol that is defined between these two parts is UDP. This protocol is used thanks to the transmission speed since there is no confirmation control in the sending and

receiving of packages.

The part that really concerns the project is the one that is included in the agent and, more specifically, to the client and in the model of the finite state machines. The client is a set of classes that implement actuators and sensors agent. As mentioned, along with the FSMs model, it is a submodule of the agent. In turn, it is composed of components that give it the necessary functionality. These components are organized by classes and interfaces and are:

- **`Action.`** This class defines the actions to be sent to the server so that the client can execute them in the game. These actions are: to accelerate, to brake, to clutch, to indicate the gear, to indicate the direction, to restart the race, to read the angles.
- **`Client.`** This is the agent's main class. It is the class responsible for establishing the connection between the client and the server, loading the driver of the vehicle and controlling the execution loop of the agent.
- **`Controller.`** This class serves as the basis for the drivers that you want to implement.
- **`SensorModel.`** This component is defined as an interface with the headers of the methods to obtain the reading of the vehicle's sensors.
- **`MessageParser.`** Through this class, the message received from the server is analyzed and divided into tokens in order to be able to access each of them individually and thus to understand in a clear way each one of the sensors of the vehicle.
- **`MessageBasedSensorModel.`** In this class, the methods of `SensorModel` are implemented based on the syntactic analysis that has been done previously in the `MessageParser` class.
- **`SocketHandler.`** This is the class that allows you to communicate with the server.

In addition, the code contains two drivers that have already been implemented:

- **`DeadSimpleSoloController.`** It is a very simple controller whose functionality is minimal.
- **`SimpleDriver.`** The complexity of the agent is much greater than `DeadSimpleSoloDriver` complexity. The controller has some final variables to have greater accuracy in the agent logic, as well as control methods of some of the sensors to adjust the value of each action to perfection.

In the figure 4 the interaction between the components of the agent is represented.

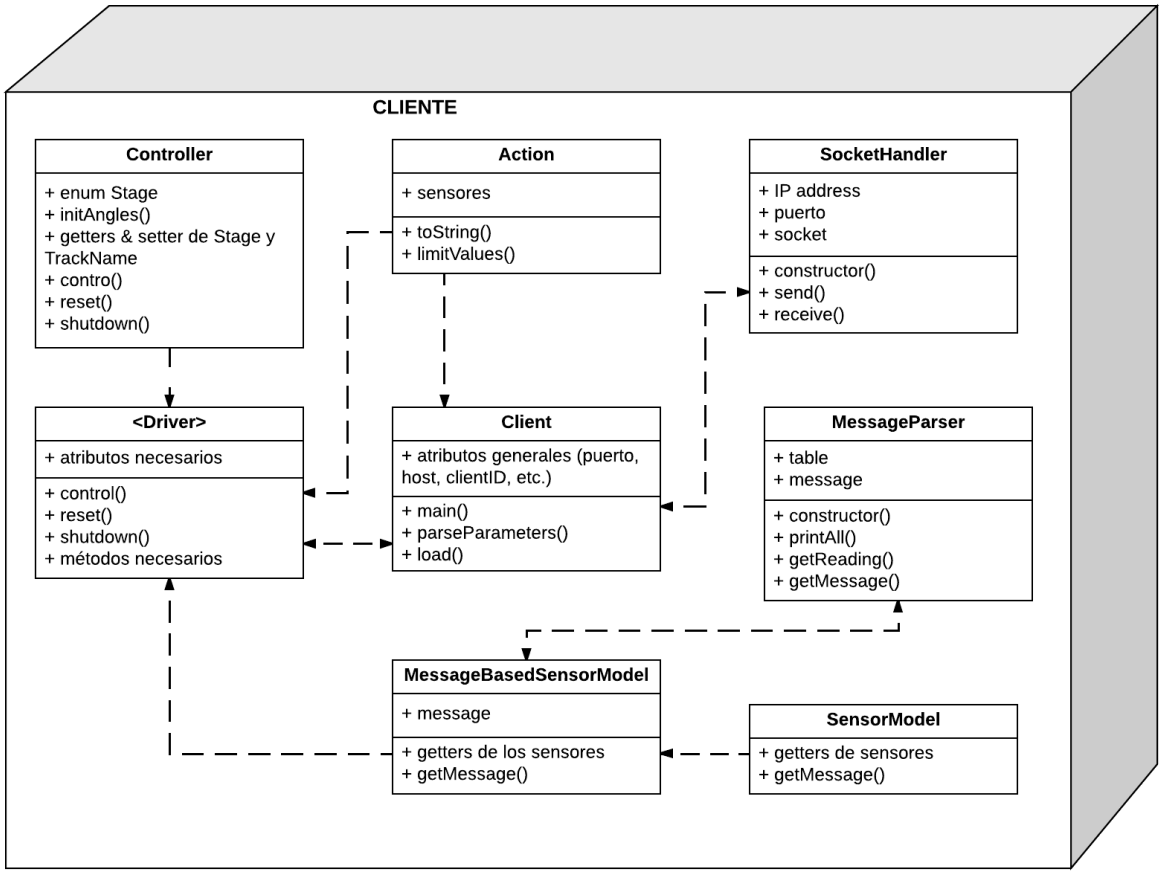

Figura 4: Component diagram (Source: own).

Also, in the figure 5 the final architecture of the system is shown without indicating the model of FSM that is used. This model is described below.

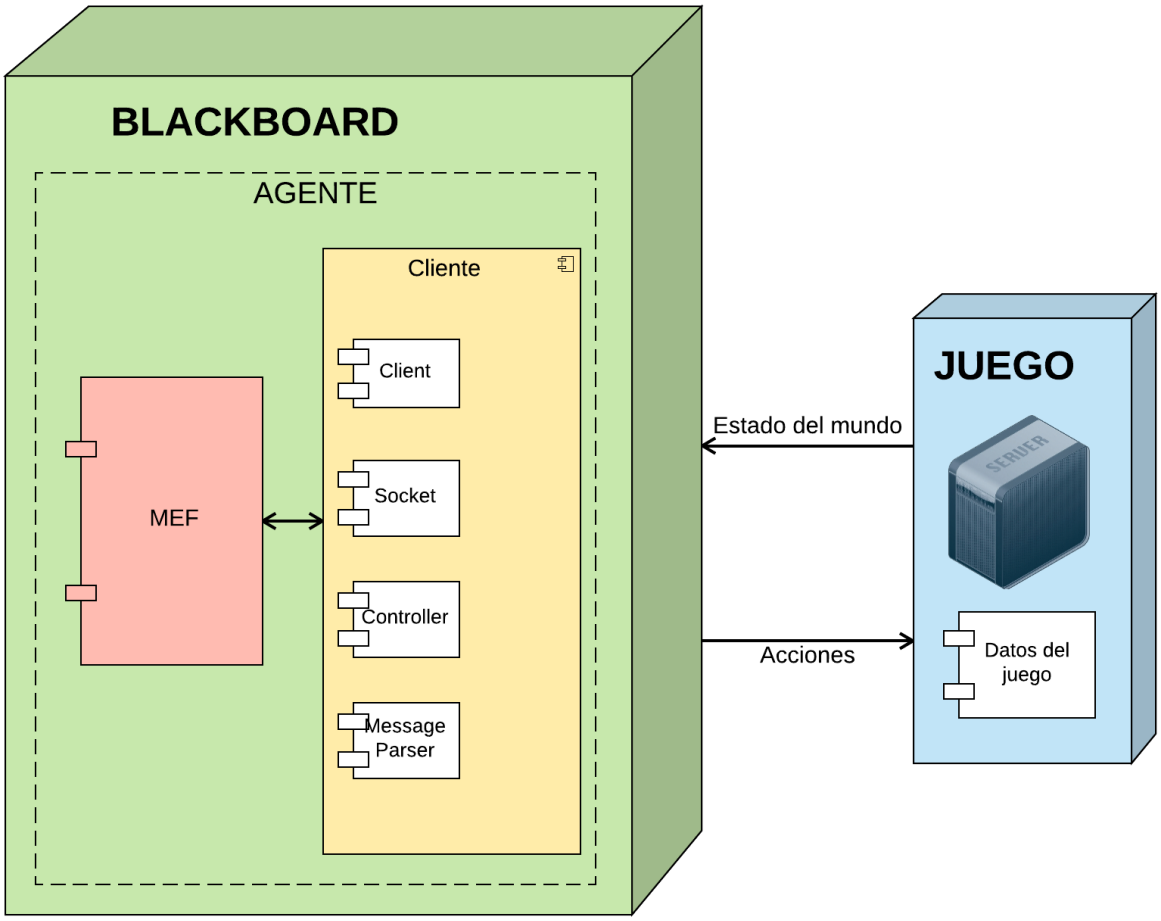

Figura 5: System's final architecture (Source: own).

In this architecture, you can see that the blackboard model has been used to group the components of the agent. The blackboard controls the coordination between the different finite state machines as well as the sensors of the vehicle and the actions that are sent to the server. In addition, it offers a communication between the FSMs and the sensors to be able to establish the transition conditions between states. This model offers a greater simplification of the system and provides consistency to it.

Regarding the FSM model, two possible solutions are proposed: a simplified model in which each state machine controls one of the actions (or two if they are directly related) and another in which there is a single machine that controls the entire system. The simplified model, which has been chosen as the final solution, follows the concept of Ockham's razor. In this way, three different finite state machines are obtained:

- **Speed control**

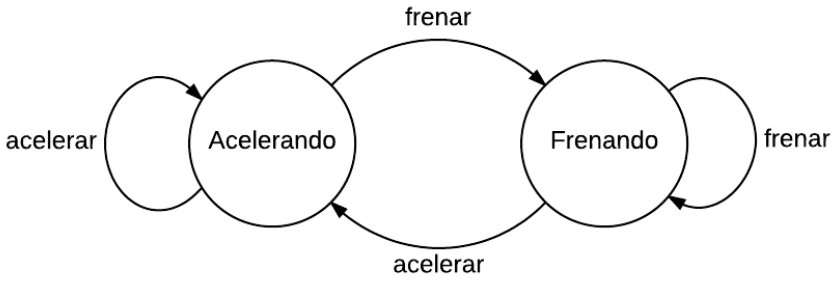

Figura 6: Speed control FSM (Source: own).

- **Gear control**

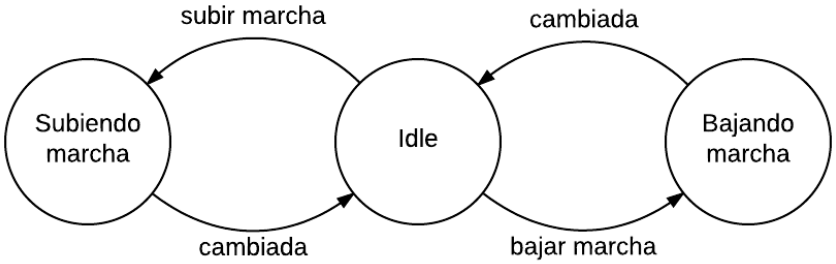

Figura 7: Gear control FSM (Source: own).

- **Steer control**

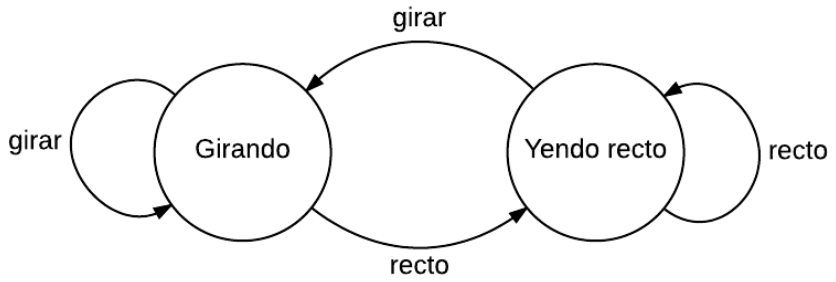

Figura 8: Steer control FSM (Source: own).

Once the system design has been defined, it is implemented. Since in Java there is no original language pack that includes state machines, it is necessary to import some external package that does or to design the necessary classes to be able to implement the FSMs. The second option has been chosen to have a class design adapted to the system. In this way, three new classes are defined and added to the rest of the code that is already available. These classes are:

- **`StateMachine.`** It is used to define state machines, to add states and, partially, control transitions.
- **`State.`** It is used to define states and transitions between them. In addition, it provides the rest of the control of the FSM transitions.
- **`ActionHandler.`** It is an interface which is used to define the header of the method that controls the action of the transition.

By means of these classes, it is possible to implement the three state machines that have been defined in the design. In the figure 9 is shown the interaction that the finite state machines have, the possible actions of the transitions between states and the control functions of the controller.

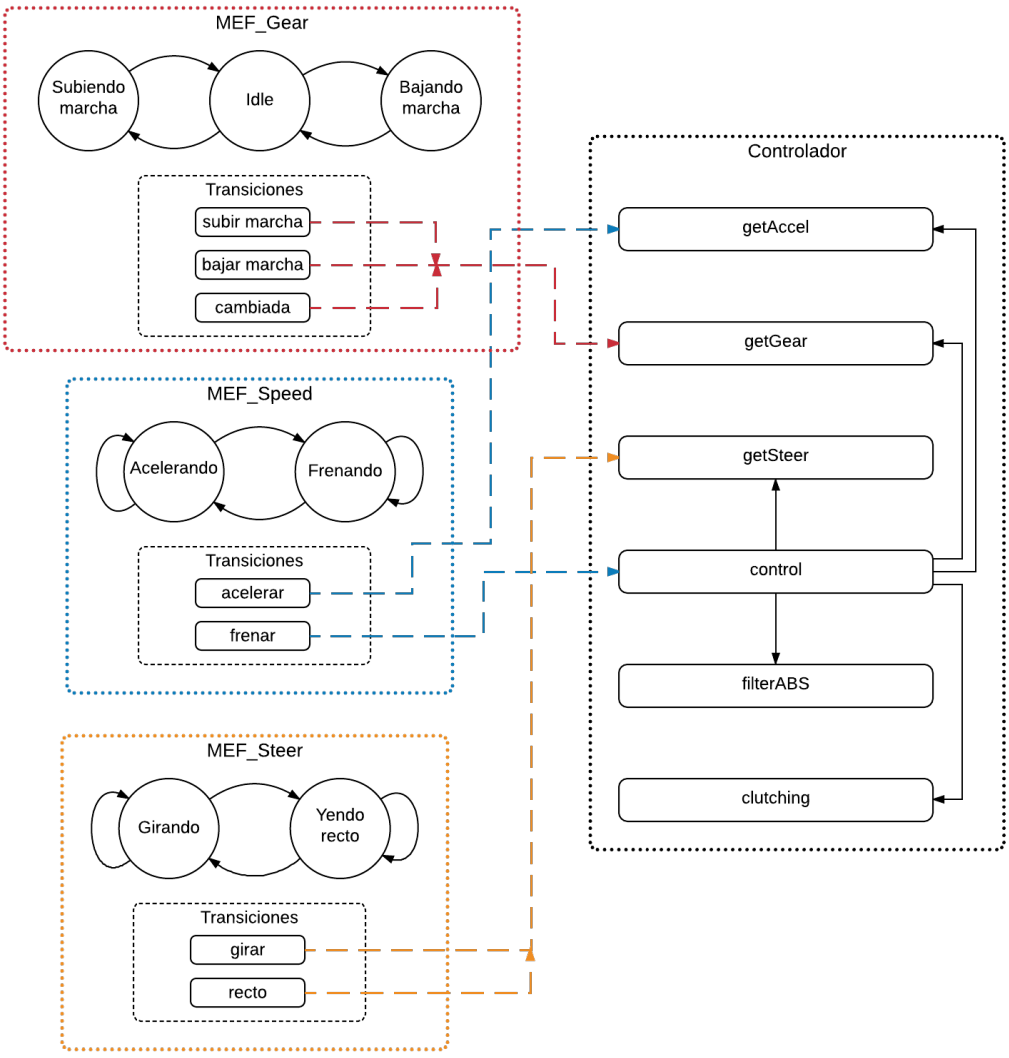

Figura 9: FSM and `SimpleDriver` controller interaction (Source: own).

After the culmination of this project, technical conclusions are drawn on it as well as some personal conclusions about the whole process.

First of all, I would like to comment on the technical conclusions that have been drawn. After completing the project, it can be stated that the main objective has been

fulfilled in the ref sec: objectives section. Through a research process, it has been possible to design and implement a finite state machine model capable of controlling an automatic agent for the TORCS. Through this model, some of the sensors of the car are controlled and managed to manage it, so that it is able to circulate along the track and even compete against some of the rivals. As it is a single main objective, the work part rests in the process of investigation and design of the FSM to find the best possible model; while the implementation part, although costly because of the amount of code to be understood and generated, is somewhat more trivial.

I would like to make some emphasis on the research process on which programming language to use. Although it has been a brief period, certain complications have arisen in with the languages of Matlab and Python, and therefore they ended up being rejected and the problem was approached with Java.

After observing the results obtained in the section of tests, it can be concluded that the solution obtained was the expected one against the objective, being possible to exist possible improvements that can be approached in future works.

As for personal conclusions, I would like to emphasize that I have just finished this chapter of my life with a certain degree of satisfaction, but bearing in mind that I might have obtained better personal results. With regard to the career, after 5 years of work, sacrifices and breaks, I have realized that you can always give more of self in the tasks. I think I could have completed my academic career with a greater degree of knowledge, but I hope that with the new chapter that I will open after this project will bring me the skills that I lack. Focusing on the project described in this document, I would like to emphasize that it has been a personal challenge. Almost two years ago I was struggling to do a similar project for one of the subjects of the course, which I could say was the one that made me suffer the most. In taking this project, I have remembered some of those moments. But, when I finish it, I feel satisfied with myself and I can settle this stage with a smile.

The development of this automatic agent is a contribution to the community which can be used in various fields. On the one hand, it can be continued with the development of it and try to refine some of its aspects, to improve the FSM that controls it or to combine it with other techniques of the artificial intelligence like the networks of neurons or the Turing machines.

The overall project can be used for the implementation of automatic agents in new video games, or at least serve as a basis for these. Changing the field, this work gives results that serve as analysis and testing prior to the development of autonomous cars.

In an academic approach, this project can be used both for teaching the operation of state machines, as well as in the development of video games.

Índice

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Desde que apareció por primera vez el término de Inteligencia Artificial (IA) en 1956, se le han adjudicado cuantiosas y variadas definiciones, sin llegar a concretar en una única. Antes de definir este término, otros científicos y pensadores ya se habían planteado la abstracción del razonamiento lógico del ser humano para poder aplicarlo a las máquinas, como fue el caso de Alan Turing y su Máquina Universal, conocida como la Máquina de Turing.

Fue en 1936 cuando el matemático británico definió y describió el funcionamiento de la Máquina de Turing (MT). Basándose en el concepto de algoritmo o proceso efectivo, Turing diseñó su máquina como un dispositivo capaz de realizar un algoritmo, apoyando las teorías de Gödel al demostrar que existen problemas que no pueden ser resueltos por una MT[1]. Gracias a las aportaciones que hizo el científico inglés, nació el Test de Turing, el cuál ha sido empleado durante años para determinar la "inteligencia" de una máquina. Será a partir de aquí desde donde parte este proyecto, pero eso se especificará más adelante.

Como se comentaba anteriormente, se ha intentado dar una definición concreta a la inteligencia artificial. A continuación se pueden encontrar algunas de las definiciones que se le ha dado al término IA, dependiendo de la aproximación que se haga de la misma[2]:

- Sistemas que piensan como humanos

  "The exciting new effort to make computers think... machines with minds, in the full and literal sense" (Haugeland, 1985)

  "El nuevo y emocionante esfuerzo de hacer a los ordenadores pensar... máquinas con mente, en el sentido completo y literal"

- Sistemas que actúan como humanos

  "The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990)

  "El arte de crear máquinas que realizan funciones que requieren inteligencia cuando son realizadas por personas"

- Sistemas que piensan de forma racional

  "The study of mental faculties through the use of computational models" (Charniak and McDermott, 1985)

  "El estudio de las facultades mentales mediante el uso de modelos computacionales"

- Sistemas que actúan de forma racional

  "The branch of computer science that is concerned with the automation of intelligent behavior" (Luger and Stubblefield, 1993)

  "La rama de la informática que se ocupa de la automatización del comportamiento inteligente"

1.1. Motivación

Con la creación de los primeros videojuegos, los desarrolladores de los mismos han buscado la forma de lograr que éstos se acerquen lo máximo posible a la realidad. Uno de los detalles principales que se requiere cuando se realiza un videojuego es conseguir que la inteligencia de los personajes que lo componen sea lo más similar posible a la de un humano, es decir, acercar el comportamiento del juego al comportamiento de la naturaleza. La IA es la encargada de lograr dicha hazaña y se podría decir que en los últimos años la progresión de ésta ha sido bastante elevada en este campo, también en parte ayudada por los avances del hardware y software que permiten un diseño más realista de los componentes de los juegos.

De forma paralela, se ha visto que algunas de las aplicaciones que se le dan a la inteligencia a son útiles para las labores de la vida cotidiana de las personas y no sólo para los videojuegos, como es el caso de los robots que se están desarrollando en los últimos tiempos (aspiradoras autónomas, ASIMO, NAO), los coches autónomos (Tesla) o los asistentes personales como Siri o Cortana. Así mismo, el desarrollo de la IA en los videojuegos, en ocasiones, es aplicable posteriormente a la realidad, lo que hace que éstos sirvan como banco de desarrollo y pruebas.

En la figura 1.1 se puede observar la evolución que ha tenido el robot ASIMO desde que se creo el primer modelo hasta el último obtenido. Por otro lado, en la figura 1.2 se puede ver un robot nao con todos los sensores que éste posee para percibir el medio que le rodea. Ambos son el fruto de años de trabajo y evolución de la IA.

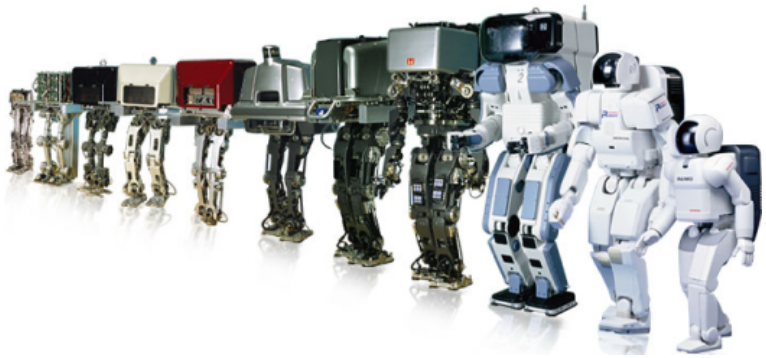

Figura 1.1: Evolución de ASIMO (Fuente: `http://elbueninfo.blogspot.com.es`).

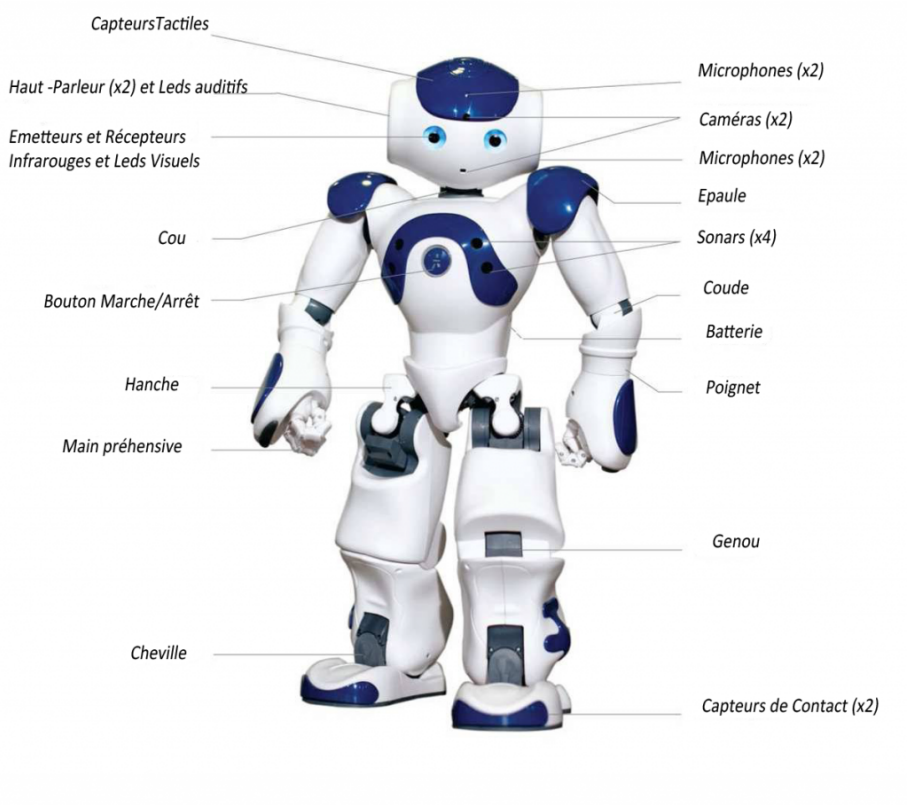

Figura 1.2: Robot NAO con sensores (Fuente: `http://www.andorobots.com`).

En el capítulo 2 se explicará de forma más detallada la situación actual tanto de los videojuegos como de las máquinas de estados y la IA en general, así como la evolución que han sufrido desde que aparecieron.

1.2. Objetivos

El objetivo principal de este proyecto es desarrollar un agente automático capaz de controlar un sistema (personaje, máquina, coche, etc.) de un videojuego. Para ello, se van a utilizar como tecnología las máquinas de estados y autómatas finitos. No se trata de la creación de una aplicación para el uso posterior de forma abierta, sino del

desarrollo y estudio de una idea, de observar cómo se pueden aplicar las máquinas de estados en este campo y compararlo con otros proyectos similares. Esto no implica que con el documento presente se indique un manual de usuario en el que se explique cómo poder realizar un trabajo parecido y cómo utilizar el código generado en este proyecto.

La elección del videojuego está sujeta a diferentes factores: licencias, complejidad, actualidad, aplicaciones, etc. En la sección 1.3 se describirá el videojuego escogido así como los motivos de dicha decisión.

También se plantea, si es posible, la participación en algún tipo de competición del agente automático que se consiga desarrollar.

1.3. Planteamiento del problema

Basándonos en lo visto anteriormente, este proyecto plantea realizar un agente para alguno de los siguientes juegos:

- Unreal Tournament 2003 (UT3)
- The Open Racing Car Simulator (TORCS)
- NERO

En un primer paso, se barajó la posibilidad de escoger el UT3 como juego, pero la licencia del mismo así como la atención que despertó en mí la idea de realizar un *bot* de un coche me hicieron cambiar de opción al TORCS.

Una vez tomada la elección del videojuego (y estudiado por encima algunos aspectos del mismo), se plantearon las diferentes formas de abordar la tecnología que se iba a utilizar para desarrollar la IA del sistema:

- Máquinas de estados
- Máquinas de Meale y Moore
- Autómatas finitos
- Autómatas a pila
- Máquinas de Turing

Se ha decidido comenzar con los modelos más simples para establecer una base y, a partir de ahí, ver cómo va evolucionando todo el sistema que se desarrolle.

En cuanto al lenguaje de programación que se va a utilizar se han planteado los siguientes:

- Java
- C

- Matlab
- Python

Se realizará un breve estudio del impacto que pueda tener cada uno en el proyecto y, a través de este, se escogerá unos de los cuatro para desarrollar el sistema en su totalidad.

1.4. Entorno operacional

En esta sección, se va a detallar el entorno operacional del proyecto, es decir, las herramientas hardware y software que han sido utilizadas para la realización del mismo.

Hardware

- Ordenador para ejecutar los programas:
  - Procesador AMD FX(tm)-8350 de 8 núcleos CPU @ 4GHz
  - Memoria RAM de 16GB
  - SO Windows 8.1 Pro de 64 bits
  - GPU NVIDIA GeForce GTX 960 2GB GDDR5

Software

- Entorno de desarrollo: *Java Eclipse Mars.2 Release (4.5.2)*
- *Java Development Kit (JDK)*: 1.8
- *Java Runtime Environment (JRE)*: 1.8
- Entorno de desarrollo: *Matlab R2016b (9.1.0.441655) 64-bit Academic License*
- Herramienta software: *Simulink*
- Herramienta software: *Git Bash (mintty 2.7.3) 86-bit*
- Software de control de versiones: *Git*
- Sistema de composición de textos: *LaTeX*
- Herramienta software de dibujo de grafos: *Draw.io* y *Lucidchart*
- Videojuego *TORCS* (1.3.4)

1.5. Estructura del documento

A continuación, se expondrá la estructura que sigue este documento con una breve descripción de cada capítulo que lo compone.

- Capítulo 1: Introducción - En este capítulo se introduce el trabajo que se ha realizado mencionando las motivaciones y objetivos que ha seguido el mismo, así como el planteamiento del problema que se ha realizado. Además se incluye el marco regulador que afecta al proyecto.
- Capítulo 2: Estado del arte - En este capítulo se aborda el estado actual de los diferentes temas que cubre el proyecto.
- Capítulo 3: Análisis del sistema - En este capítulo se describirán los requisitos que el sistema debe cumplir así como los casos de uso que cubre.
- Capítulo 4: Arquitectura y diseño del sistema - En este capítulo se definirá e ilustrará la arquitectura del sistema así como los módulos que lo componen.
- Capítulo 5: Implementación del sistema - En este capítulo se comentará la implementación que se ha llevado a cabo del sistema para resolver el problema planteado.
- Capítulo 6: Resultados y evaluación - En este capítulo se muestran las pruebas y resultados obtenidos con la implementación mencionada en el capítulo anterior.
- Capítulo 7: Planificación del trabajo, presupuesto y entorno socio-económico - En este capítulo se hará mención a la planificación del proyecto así como al presupuesto y el entorno socio-económico al que afecta.
- Capítulo 8: Conclusiones y trabajos futuros - En este capítulo se explican las conclusiones obtenidas tras realizar este proyecto así como los trabajos futuros que puede partir a través de éste.

Al final del documento se encuentra la bibliografía que se ha utilizado para realizar el mismo, así como un anexo con un manual de usuario sobre el proyecto.

1.6. Marco regulador

Dado que este proyecto de fin de grado está basado en la utilización de un videojuego, el trabajo debe atenerse a la licencia y derechos de autor (©) del juego indicado. En este caso se está trabajando sobre TORCS, un videojuego cuyo código está registrado bajo la Licencia Pública General de GNU[3], mejor conocida como GPL (siglas en inglés), la cual es una de las licencias más utilizadas en software libre y código abierto. La versión actual de la licencia es la 3.0 y, al igual que TORCS, el proyecto completo adopta dicha licencia y versiones posteriores y es bajo la cuál se rige.

También cabe mencionar que los fragmentos de código utilizados en la parte del cliente del software están bajo esta misma licencia como se indica en los comentarios de los archivos o en las páginas web desde donde han sido descargados.

Capítulo 2

Estado del arte

En este capítulo de la memoria se va a comentar el estado actual de todos los temas que engloban el proyecto: agentes automáticos, máquinas de estados, videojuegos y, en concreto, el TORCS. En el apartado de videojuegos también se hará mención a los simuladores de conducción, puesto que el videojuego que se utiliza en este proyecto se trata como tal, además de que el desarrollo que toma el mismo tiene como base, en cierto modo, a los simuladores.

2.1. Máquinas de estados y autómatas finitos

Como se ha comentado anteriormente, existen diferentes tipos de máquinas de estados, así que en este punto del capítulo 2 se mencionará la actualidad en general de éstas.

Una máquina de estados, también conocido como autómata finito, es una entidad que está compuesta por uno o más estados y una o más transiciones, siempre finitos. Los estados representan la situación en la que se encuentra el sistema que controlan, mientras que las transiciones son el reflejo de una acción que ha afectado al sistema, tanto si la ha realizado él mismo como si tiene un origen externo. En algunas ocasiones, la transición puede generar una salida que pueda afectar al funcionamiento de la máquina en acciones posteriores, así como modificar el entorno con las acciones que lleve acabo el sistema tras dicha transición.

Para conocer en profundidad qué es una máquina de estados (MS) o autómata finito (AF), primero es importante definir los tipos de AF que nos podemos encontrar.

Tipos

Los AF se pueden clasificar en dos tipos:

- **Deterministas:** cada combinación (estado + transición) produce un solo estado.
- **No Deterministas:** cada combinación (estado + transición) produce 1 o más estados. Además se permiten las transiciones con el palabra o símbolo vacío (λ).

También pueden ser representados de dos formas diferentes:

- **Diagramas de transición:** diagrama que comprende los estados y transiciones de la siguiente manera:

- Estados: se representan con nodos y la nominación formal es $\mathtt{q}_i$ donde $\mathtt{q}_i \in$ conjunto de estados ($\mathtt{Q}$).
- Transiciones: se representan mediante arcos entre nodos (estados) si existe una transición entre $\mathtt{q}_i$ y $\mathtt{q}_j$. La nominación formal es $\mathtt{e}_i$.
- Estado inicial: se representa señalándolo con $\rightarrow$.
- Esado meta o final: se representa señalándolo con $*$ o con doble círculo.

En la figura 2.1 se puede observar la representación de un autómata finito genérico mediante un diagrama de transición. El autómata está compuesto por 3 estados (1 solo estado final) y transiciones entre ellos.

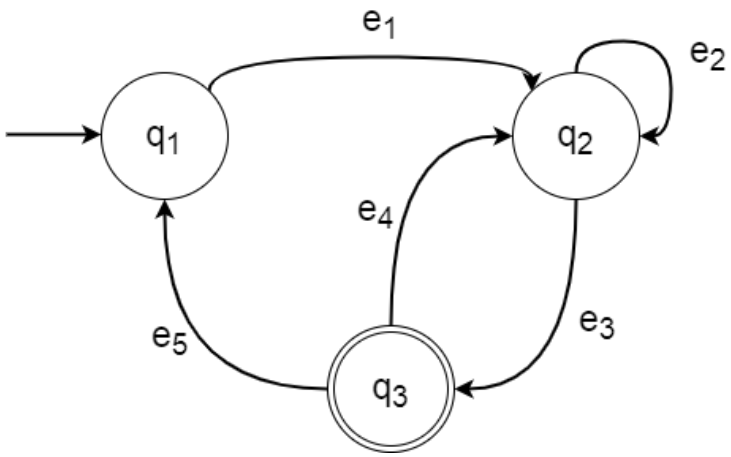

Figura 2.1: Representación de AF con diagrama de transición (Fuente: elaboración propia).

- **Tablas de transición:** tabla que comprende los estados y transiciones de la siguiente manera:
  - Estados: encabezan las filas de la tabla. La nominación formal es $\mathtt{q}_i$ donde $\mathtt{q}_i \in$ conjunto de estados ($\mathtt{Q}$).
  - Transiciones: encabezan las columnas de la tabla. La nominación formal es $\mathtt{e}_i$.

  En el cuadro 2.1 se puede apreciar la representación de un autómata finito genérico mediante una tabla de transición. A diferencia del caso anterior, en este no se ha definido un número exacto de estados ni transiciones (a pesar de que siempre tiene que ser un número finito) para que se pueda observar que tanto el número de estados y transiciones es variable a la hora de definir el AF.

Tabla 2.1: Representación de AF con tabla de transición

	e_1	e_2	...	e_m
q_1	$f(q_1, e_1)$	...	...	...
...	...	...	...	...
q_n	...	...	...	$f(q_n, e_m))$

Autómatas Finitos Deterministas

Un autómata finito determinista (AFD) se define a través de un quíntupla:

$$(\Sigma,\ Q,\ f,\ q_0,\ F)$$

- Σ: alfabeto de entrada; caracteres que son reconocibles por el autómata.
- Q: conjunto de estados; es un conjunto finito no vacío, es decir, al menos hay un estado y siempre con número de estados definido. Se utiliza como un alfabeto para distinguir a los estados.
- f: función de transición; también se puede representar como:

$$Q x \Sigma \rightarrow Q$$

- q_0: estado inicial; $q_0 \in Q$.
- F: conjunto de estados finales o de aceptación; $F \subset Q$.

A continuación se muestra un ejemplo de un AFD:

$$AFD = (\{0,\ 1\},\ \{p,\ q,\ r\},\ f,\ p,\ \{q\})$$

donde f está definida por:

f(p,0)=q	f(p,1)=r
f(q,0)=q	f(q,1)=r
f(r,0)=r	f(r,1)=r

En la figura 2.2 se aprecia la representación del ejemplo de AFD, que se ha descrito anteriormente, con una tabla de transición así como su diagrama de transición equivalente. Se pueden observar los 3 estados que componen el autómata (p, q, r) así como las transiciones que hay entre ellos y el alfabeto de entrada que es reconocido por el mismo.

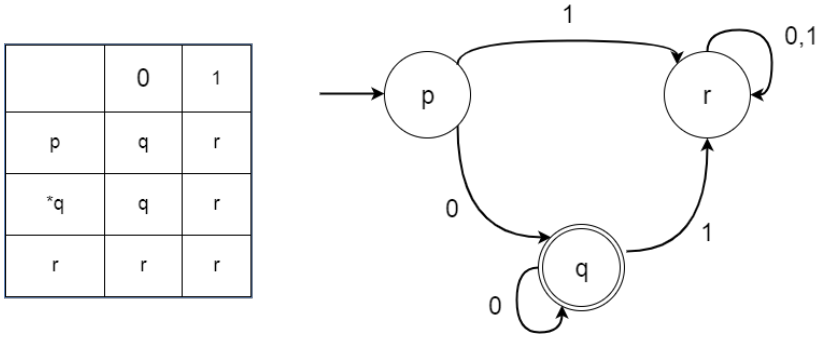

Figura 2.2: Ejemplo de AFD con tabla y diagrama de transición (Fuente: elaboración propia).

De este ejemplo se pueden destacar los siguientes aspectos:

- `p` es el estado inicial. Cualquier palabra que se quiera intentar reconocer mediante este autómata empezará sus transiciones a partir de este estado;
- `q` es el estado final o meta. Cualquier palabra que alcance este estado al final de su cadena se puede decir que ha sido aceptada por el autómata;
- este autómata sólo reconoce como elementos del lenguaje el 0 y el 1;
- este autómata solo acepta como palabras cualquier cadena de 0's. En cuanto llega un 1 el autómata transita al estado `r` del cual no se puede salir.

Autómatas Finitos No Deterministas

Al igual que el AFD, un autómata finito no determinista (AFND) se define mediante una quíntupla:

$$(\Sigma,\ Q,\ f,\ q_0,\ F)$$

con la diferencia que ahora la función de transición sí acepta la palabra vacía:

$$Qx(\Sigma \cup \lambda) \rightarrow Q$$

Un ejemplo de AFND es el siguiente:

$$AFND = (\{a,\ b\},\ \{p,\ q,\ r,\ s\},\ f,\ p,\ \{p,\ s\})$$

donde f está definida por:

$$f(p,a)=q \qquad f(q,\lambda)=s$$
$$f(q,a)=p,r,s \qquad f(q,b)=p,r$$
$$f(r,\lambda)=r,s \qquad f(r,b)=p,s$$
$$f(s,\lambda)=s$$

En la figura 2.3 se puede obserbar la tabla de transiciones así como el diagrama de transiciones del AFND anterior.

	a	b	λ
→*p	q		
q	p,r,s	p,r	s
r		p,s	r,s
*s			s

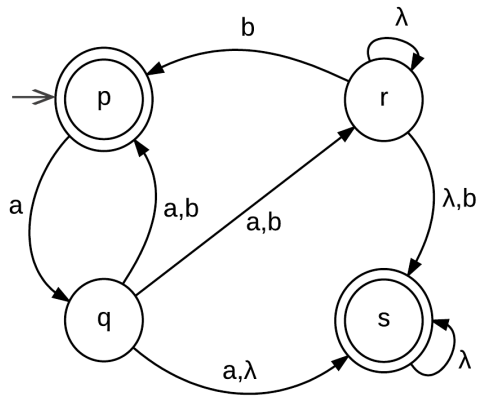

Figura 2.3: Ejemplo de AFND con tabla y diagrama de transición (Fuente: elaboración propia).

Equivalencia entre AFD y AFND

Dado un AFND siempre es posible encontrar un AFD equivalente, es decir, que reconozca el mismo lenguaje. Un AFND no es más potente que un AFD, sino que un AFD es un caso particular de AFD. Esto no implica que un AFND sea capaz de resolver más problemas que un AFD, sino que ambos tipos de autómatas finitos son capaces de resolver el mismo tipo de problemas.

Otras máquinas de estados

A parte de los tipos de autómata finito que se han visto en los apartados anteriores, existen otros tipos de máquinas de estados con mayor capacidad de cómputo debido a las diferentes características de cada una:

- **Autómata probabilístico o modelos ocultos de Markov:** fueron los precursores de las máquinas de estados. Su nivel de cómputo es muy similar al de éstas. Se diferencia de una MEF en que en sus transiciones existe cierta probabilidad, lo que implica que no siempre con la misma entrada se tome la misma transición partiendo del mismo estado. En la figura 2.4 se observa un ejemplo de Modelo oculto de Markov.

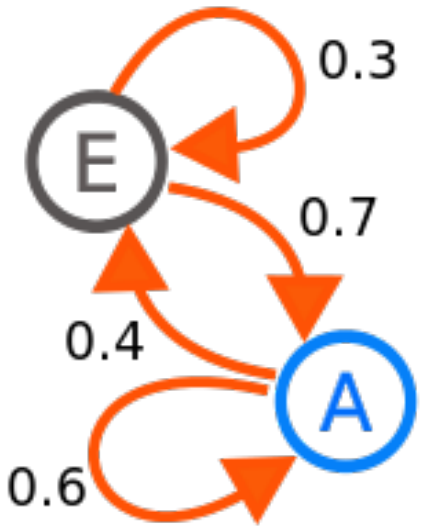

Figura 2.4: Autómata probabilístico o Modelo oculto de Markov (Fuente: `https://es.wikipedia.org`).

- **Autómata a pila:** es bastante similar a un AFD, pero disponen de una pila en la que se puede "guardar" cierta información que le dota de una mayor capacidad de cómputo. La simbología de las transiciones es un poco más compleja que la de un AF ya que se debe indicar si se apila o desapila y, en el caso de que se tenga que apilar algún elemento, hay que indicar qué elemento se apila. La figura 2.5 muestra un ejemplo de un autómata a pila.

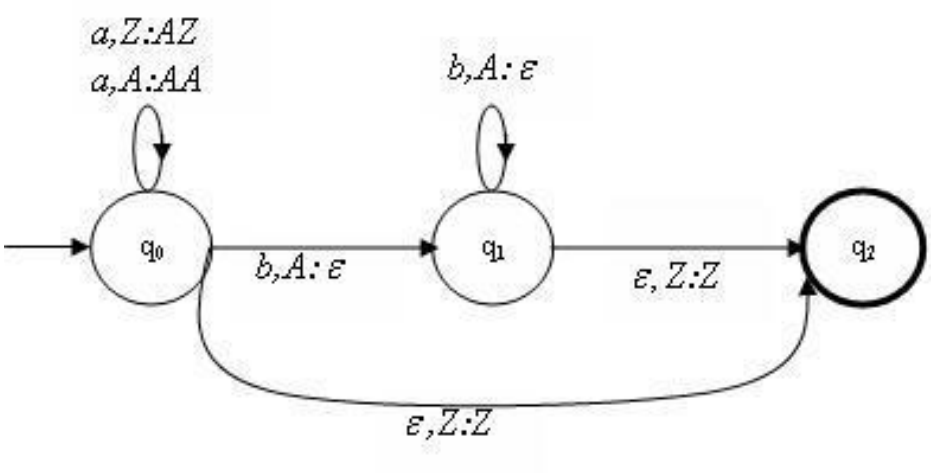

Figura 2.5: Autómata a pila (Fuente: `https://es.wikipedia.org`).

- **Maquina de Turing:** como ya se mencionaba en el capítulo 1, la Maquina de Turing (MT) tiene una alta capacidad de cómputo. Un MT lee caracteres o símbolos de una cinta dependiendo de cómo se haya diseñado la máquina. Las Máquinas de Turing tiene diferentes variaciones que la dotan de nuevas características. En la figura 2.6 se puede ver una representación de una Maquina de Turing.

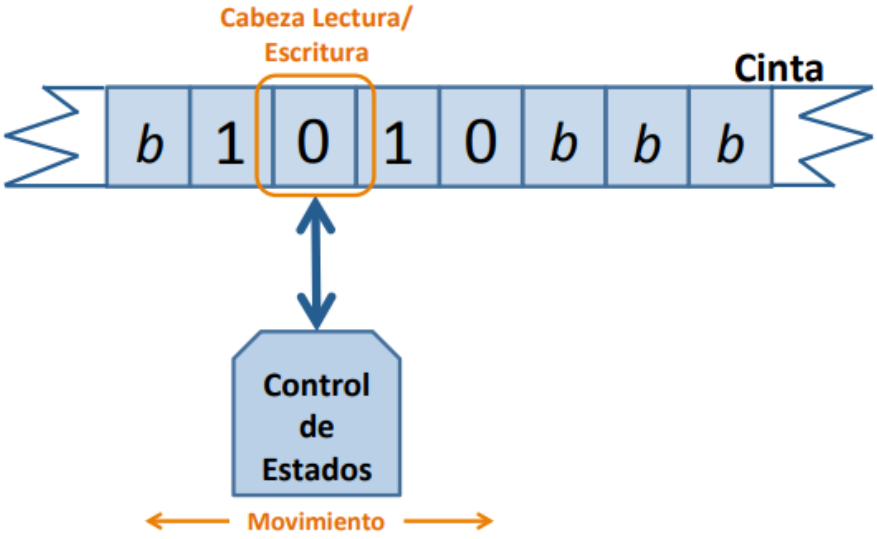

Figura 2.6: Máquina de Turing (Fuente: elaboración propia).

- **Autómatas celulares:** tal como dice David Alejandro Reyes Gómez "Un autómata celular es un modelo matemático para un sistema dinámico, compuesto por un conjunto de celdas o células que adquieren distintos estados o valores. Estos estados son alterados de un instante a otro en unidades de tiempo discreto, es decir, que se puede cuantificar con valores enteros a intervalos regulares. De esta manera este conjunto de células logran una evolución según una determinada expresión matemática, que es sensible a los estados de las células vecinas, la cual se le conoce como regla de transición local."[4]. La figura 2.7 muestra 2 ejemplos

diferentes de autómata celular: a la izquierda regla 90 de Wolfram y a la derecha un autómata celular en 3D.

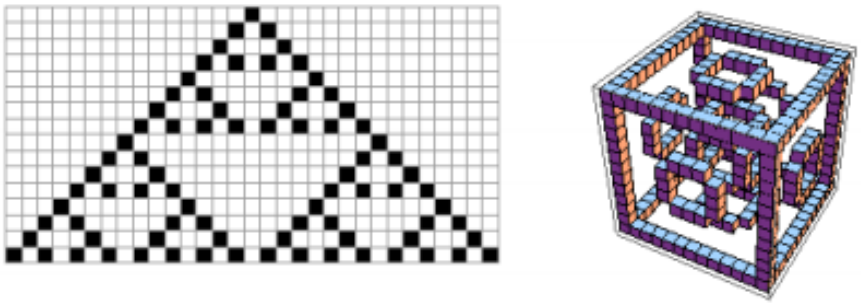

Figura 2.7: Autómatas celulares (Fuente: `http://uncomp.uwe.ac.uk/genaro`).

- **Red de neuronas:** se trata de la unión de varias neuronas artificiales que tratan de imitar el comportamiento y la funcionalidad de las neuronas biológicas. La capacidad de cómputo es bastante elevada debido a aptitud de aprendizaje que poseen. En la figura 2.8 se observa una representación genérica de una red de neuronas.

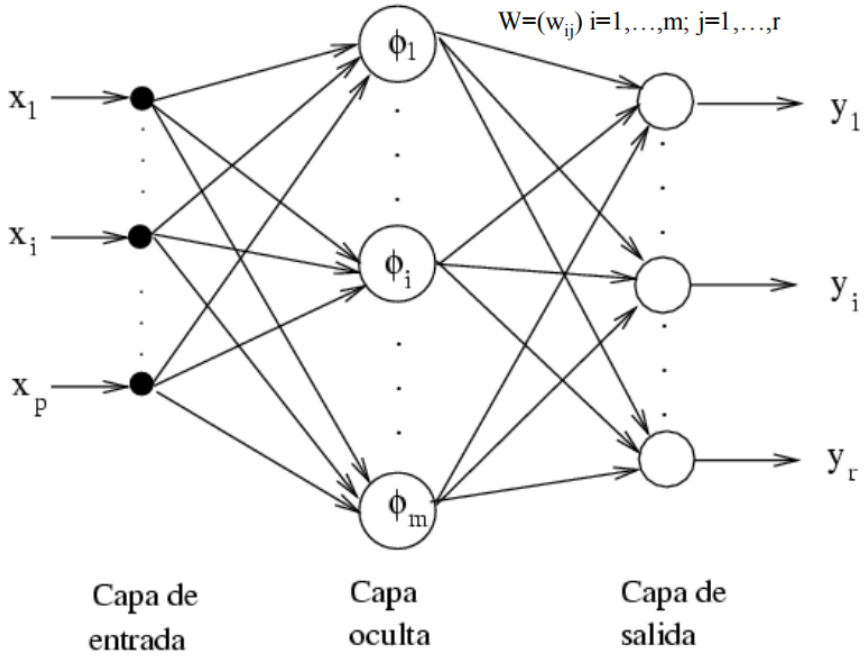

Figura 2.8: Red de neuronas (Fuente: elaboración propia).

2.1.1. Máquinas de estados en sistemas eléctricos y electromecánicos

Con la aparición de los sistemas eléctricos se requería un control lógico del mismo. Las máquinas de estados son una tecnología "básica" que puede resultar muy útil para este tipo de controles y manejar de forma segura diferentes sistemas eléctricos que son muy básicos como semáforos, ascensores o máquinas expendedoras. Es en los controladores de estos sistemas donde se implementan las máquinas de estados (ME). Muchas

de las máquinas de hoy en día tienen circuitos integrados (con millones de transistores) para los cuales se programa la lógica que lleva a cabo la tarea de control.

Los circuitos integrados presentan ciertas limitaciones, pero como se tratan de sistemas con tareas muy sencillas, los únicos inconvenientes que puedan surgir son mecánicos (industriales). En los siguientes apartados se presentan algunos de estos sistemas que incluyen ME en sus controladores.

Semáforos

Este es uno de los casos donde la lógica condicional que se aplica debe tener en cuenta la presencia de diferentes variables:

- número de semáforos que controlan el tráfico a la vez;
- si es peatonal o no;
- el tiempo de duración de cada estado.

Esta última variable es muy importante e implica un elemento adicional en el sistema: un temporizador. Mediante un temporizador se introduce la señal de tiempo a la máquina de estados la cuál toma las decisiones en función del valor que le llega. Por su parte, la ME debe devolver una señal al temporizador para ajustar el nuevo valor de tiempo dependiendo del estado al que se transite.

Por otro lado, cuantos más semáforos sea necesario sincronizar más estados son necesarios en la ME. Además, si se añade que son semáforos peatonales, se necesitan estados para controlar este semáforo adicional así como un temporizador para cada uno.

En la figura 2.9 se puede observar un controlador simple de un semáforo sin la parte del semáforo peatonal. Por un lado está el temporizador y por otro la MEF que controla la lógica. En la figura 2.10 se muestra el diseño del diagrama de transición de la ME.

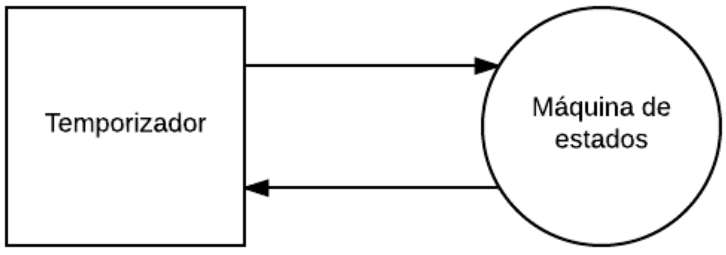

Figura 2.9: Ejemplo del controlador de un semáforo (Fuente: elaboración propia).

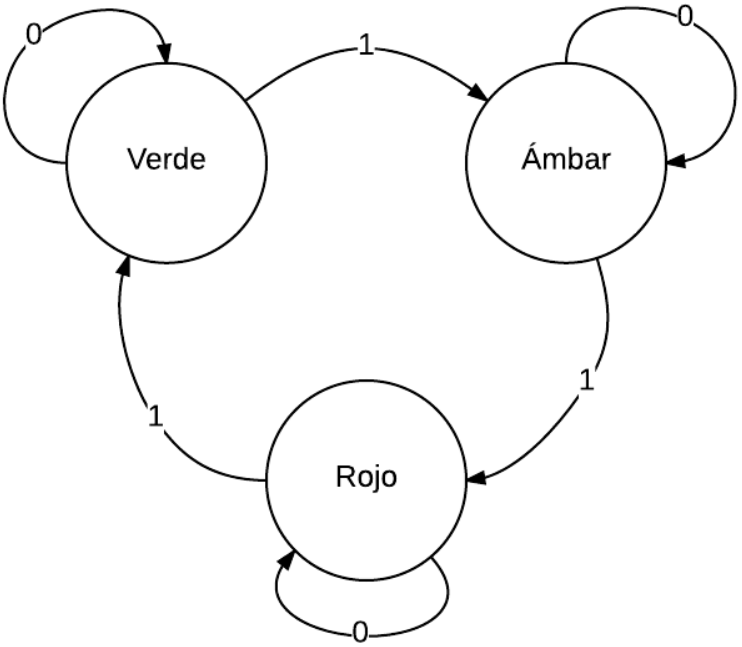

Figura 2.10: Ejemplo MEF de un semáforo (Fuente: elaboración propia).

Si se analiza el diagrama de transición se puede entender que 1 significa que ha llegado la señal del temporizador, mientras que el 0 significa que sigue esperando la señal. Las transiciones con 0 podrían eliminarse y limitar la MEF a recibir sólo las señales del temporizador cuando el valor es 0 para éste. Las imágenes mostradas se tratan de ejemplos simples para comprender lo que se menciona en este apartado, pero existen controladores y máquinas de estados mucho más complejas para sincronizar semáforos.

Un ejemplo de esto sería un problema en el que haya que diseñar ambos componentes para controlar 2 semáforos simultáneos.

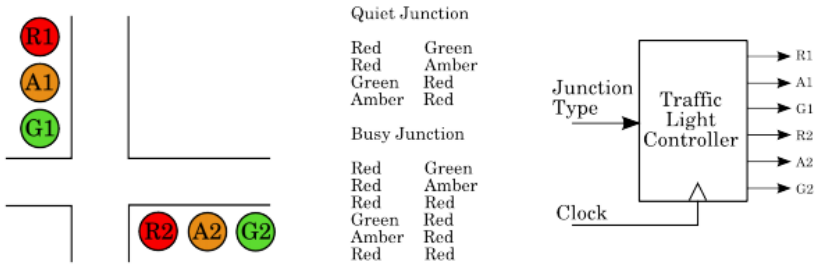

Figura 2.11: Planteamiento de un problema de semáforos (Fuente: `http://www.doc.ic.ac.uk/~dfg/`).

El profesor del Imperial College de Londres Duncan Fyfe Gillies, recoge en una de sus lecturas un problema de este tipo en el que explica todos los pasos a seguir para obtener el controlador deseado, desde el planteamiento del problema hasta alcanzar la solución final[5]. En la figura 2.11 se observa el planteamiento del problema mientras que en la figura 2.12 se puede apreciar la solución que se alcanza en el caso mencionado anteriormente.

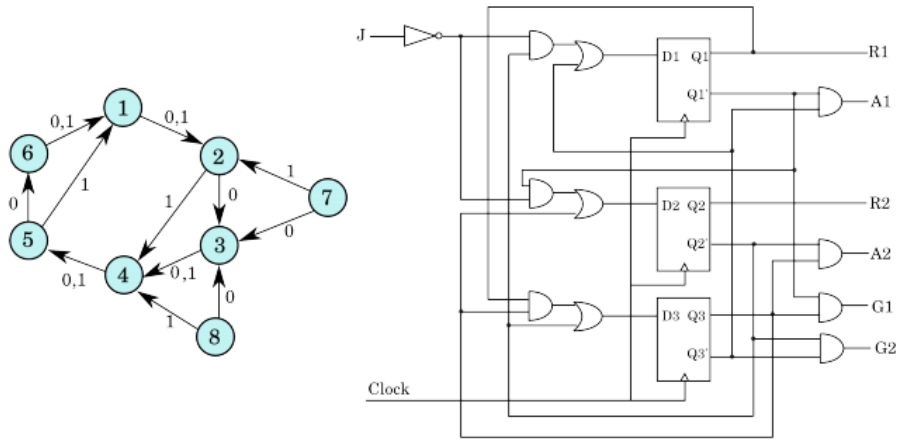

Figura 2.12: Controlador + MEF de un semáforo complejo (Fuente: `http://www.doc.ic.ac.uk/~dfg/`).

Máquinas expendedoras

Si nos preguntamos cómo calcula una máquina expendedora el dinero que se ha introducido para llevar la cuenta de éste y saber si hay dinero suficiente, si es necesario devolver y cuánto hay que devolver, podemos toparnos con las máquinas de estados. Una vez más se trata de un sistema eléctrico (incluso podría llegar a ser electromecánico si se utilizan componentes para calcular el peso de las monedas en vez de el tamaño) que necesita ser dotado de cierta lógica para poder funcionar correctamente.

La figura 2.13 representa un AFD que describe el comportamiento de una máquina expendedora la cuál solo acepta monedas de 1€ y 25 céntimos. Una bebida en esta máquina cuesta 1'25€ y sólo se podrá seleccionar el refresco deseado si se ha introducido esa cantidad o un número mayor de dinero. Si se intenta seleccionar una bebida sin haber alcanzado un estado final, el AFD no transita de estado. Este AFD se puede definir como:

$$\texttt{AFD} = (\{\texttt{0'25€, 1€, } \textit{select}^{1}\}, \{\texttt{0€, 0'25€, 0'5€, 0'75€, 1€, 1'25€, 1'5€, 1'75€, 2€}^{2}\}, \texttt{f, 0€, } \{\texttt{1'25€, 1'5€, 1'75€, 2€}\})$$

donde f está definida por el diagrama de transiciones:

[1] Tanto los estados como los símbolos del lenguaje se pueden codificar con letras y números. En este caso no se hace para que se vea de forma gráfica

[2] Se limita a un máximo de 2€ pero se puede ajustar con alguna cantidad mayor o entendiendo este estado como mayor o igual a 2€.

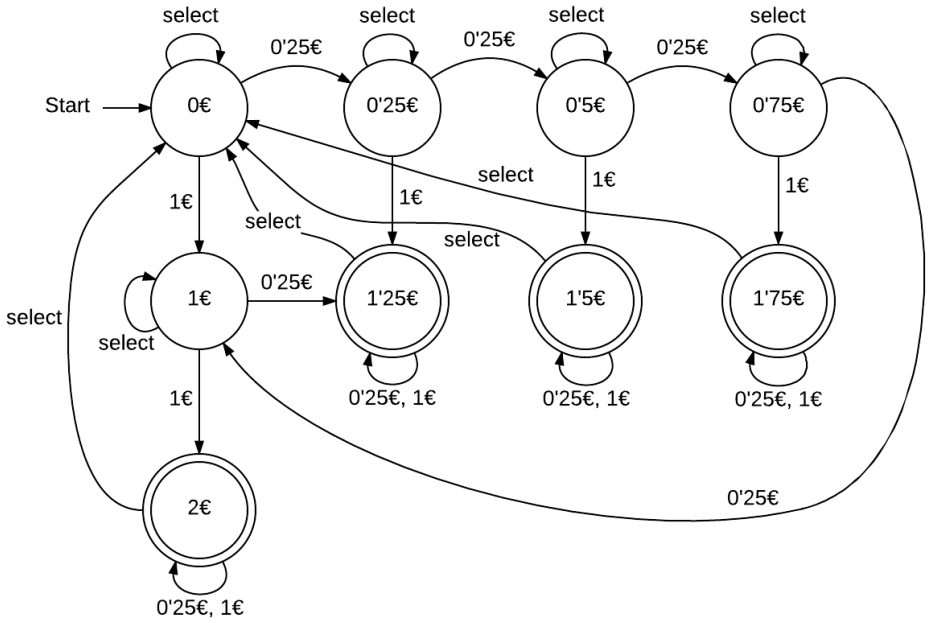

Figura 2.13: Diagrama de transiciones del AFD de una máquina expendedora (Fuente: elaboración propia).

Las máquinas expendedoras son una de las principales aplicaciones de las máquinas de estados dentro de los sistemas electromecánicos. Esto se puede apreciar en el trabajo de investigación de Ana Monga y Balwinder Singh [6] o en el de Ashwag Alrehily, Ruqiah Fallatah y Vijey Thayananthan [7]. Ambos trabajos son similares y en ellos se explica el proceso de completo del desarrollo de una máquina de estados o autómata finito determinista para un caso en particular. Mientras que Monga y Singh realizan su trabajo a cercar de una máquina expendedora cualquiera con Maquinas de Mealy, Alrehily, Fallatha y Thayananthan se centran en una máquina expendedora de libros con una MEF. En ambos trabajos se menciona que la parte del diseño de la máquina de estados es crucial para poder obtener un sistema poco complejo y que no encarezca el hardware. En las figuras 2.14 y 2.15 se puede observar los resultados finales que obtienen en ambos trabajos.

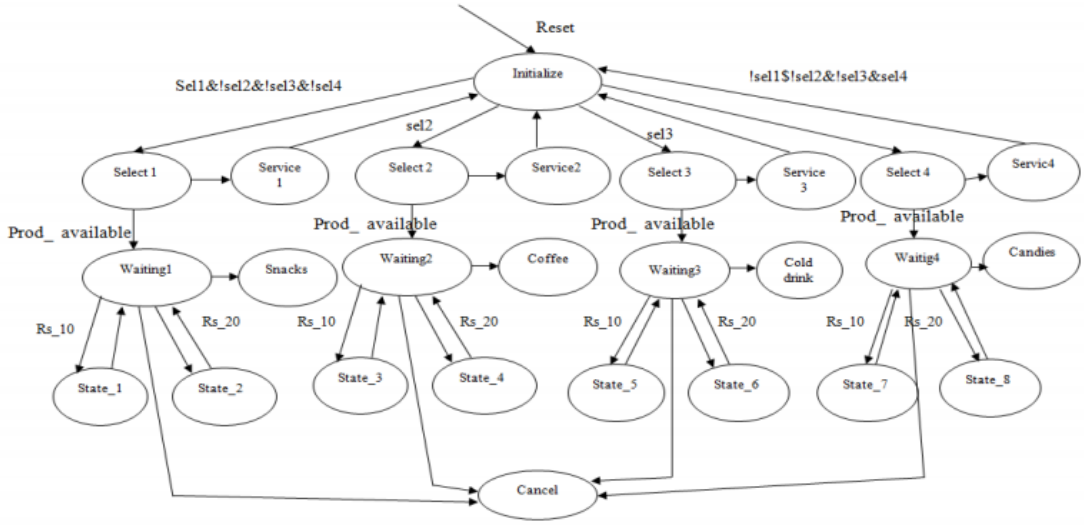

Figura 2.14: Máquina de Mealy de la solución de Monga y Singh (Fuente: `http://aircconline.com`).

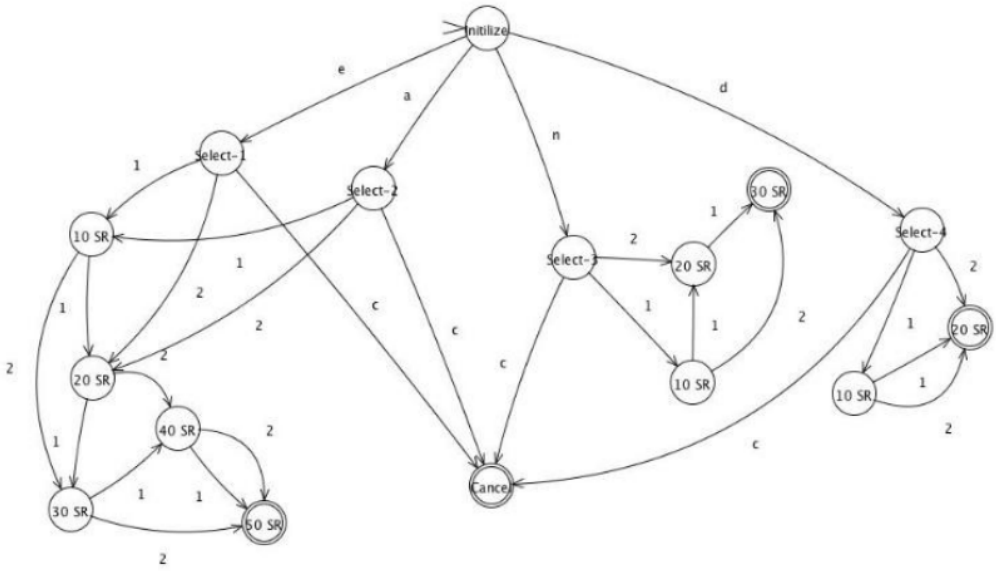

Figura 2.15: MEF de la solución de Alrehily, Fallatah y Thayananthan (Fuente: `https://www.researchgate.net`).

Otros

Como se ha visto en los apartados anteriores, las máquinas de estados son una solución para aplicar la lógica de control de muchos sistemas eléctricos y electromecánicos. A parte de los que ya se han visto, otros ejemplos son los electrodomésticos como lavadoras, microhondas o lavavajillas, ascensores, componentes de las computadoras o relojes. En todos ellos se puede encontrar algún elemento que requiera cierto control mediante una lógica.

En las figuras 2.16 y 2.17 se pueden observar dos ejemplos diferentes de cómo definir un diagrama de transiciones para la máquina de estados que controla el sistema de un ascensor; mientras que la figura 2.18 muestra la máquina de estados del control programable de una lavadora[8].

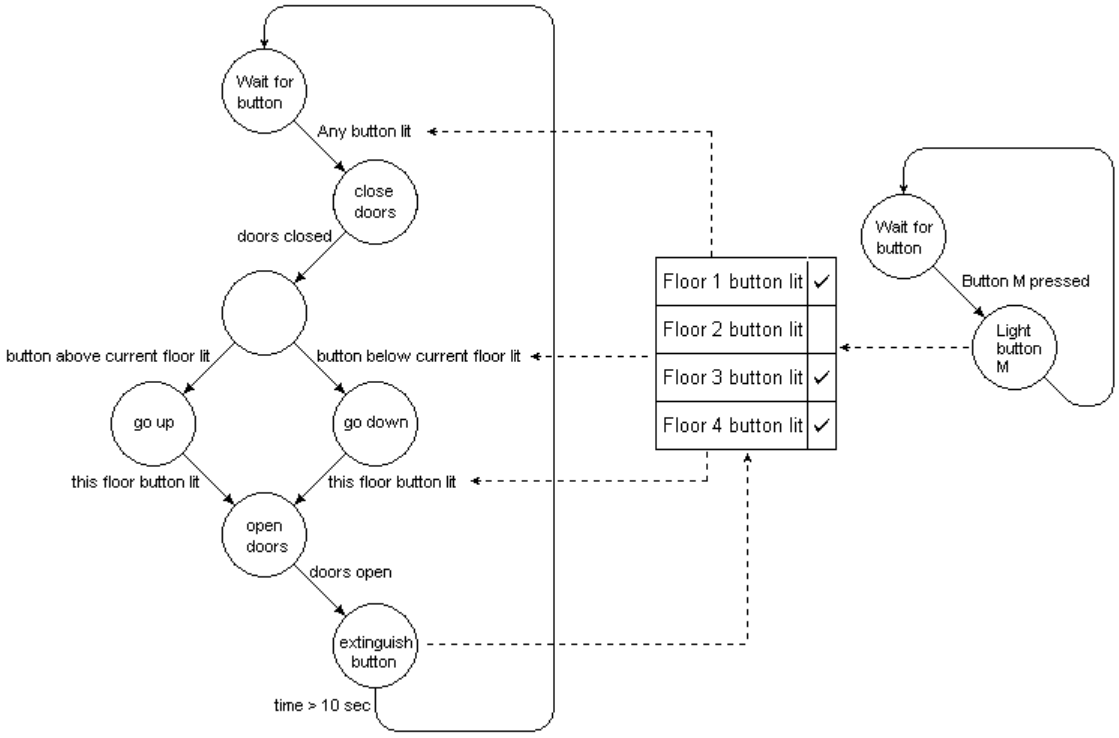

Figura 2.16: Diagrama de transiciones de un ascensor (Fuente: `http://www.peterbalch.co.uk/behaviours.htm`).

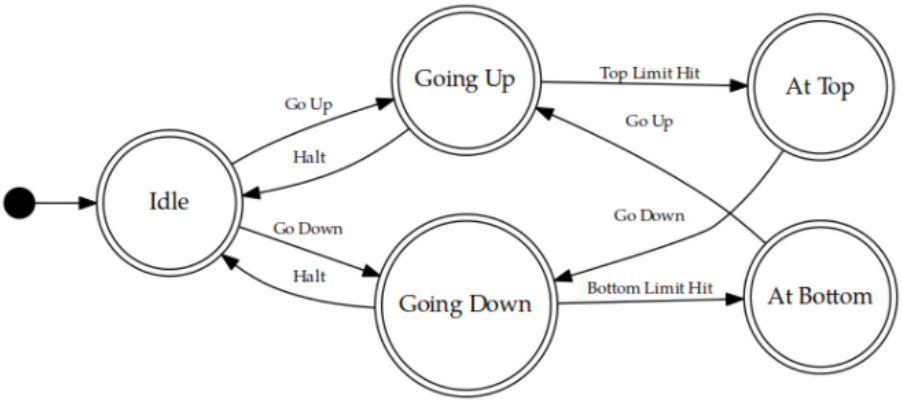

Figura 2.17: Máquina de estados de un ascensor (Fuente: `https://codereview.stackexchange.com`).

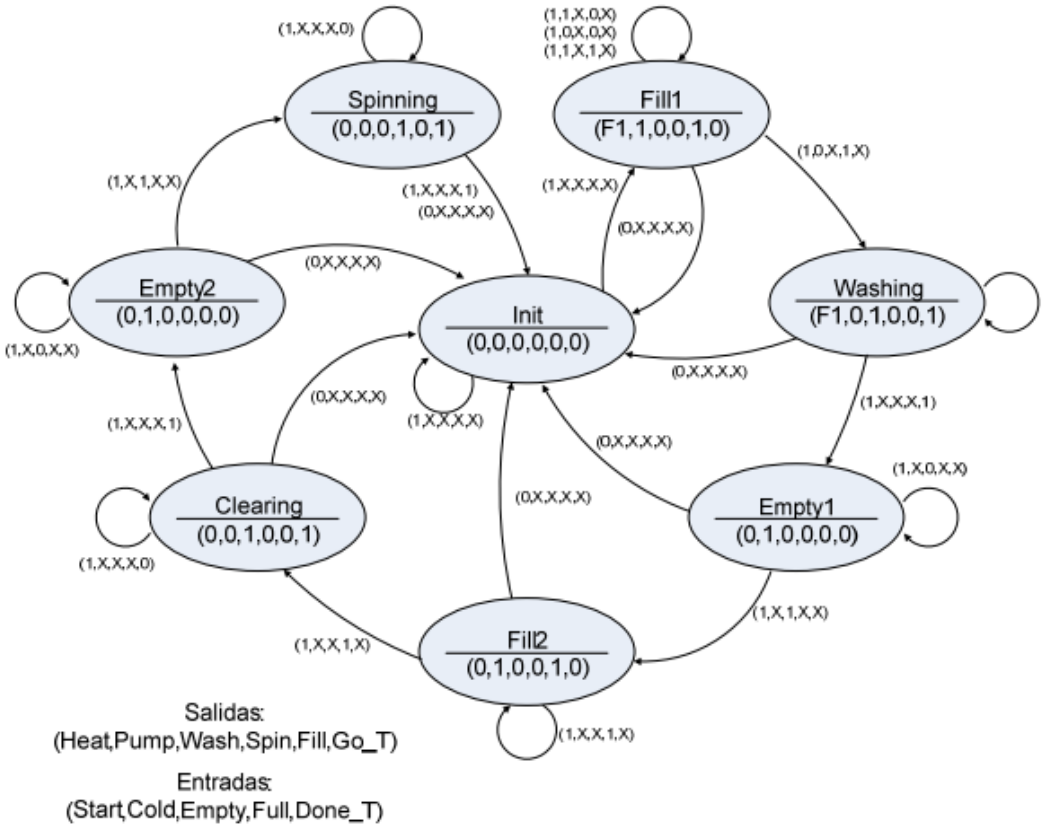

Figura 2.18: Máquina de estados del control programable de una lavadora (Fuente: `http://cc.etsii.ull.es`).

2.1.2. Máquinas de estados en reconocimiento del lenguaje natural

Fuera del campo de la electrónica y la mecánica, las máquinas de estados tienen grandes aplicaciones. Dando un giro de 180◦ grados, adentrándonos en el mundo de la lingüística (en principio nada que ver con la informática) podemos encontrar algunas de las aplicaciones más importantes de las máquinas de estados. Si unimos el campo de las lenguas con el de la inteligencia artificial obtenemos como resultados el procesamiento del lenguaje natural (PLN).

Procesamiento del lenguaje natural

El procesamiento del lenguaje natural (PLN) es un campo de la inteligencia artificial, la computación y la lingüística cuyo objetivo es hacer posible la interacción entre máquinas (computadoras) y lenguaje humano. Mediante el PLN se trata de conseguir mecanismos eficaces computacionalmente. Algunas de las tareas de trabajo del PLN son:

- **Reconocimiento del habla:** es uno de los principales campos del PLN. Se lleva trabajando en este campo mucho tiempo puesto que resulta bastante útil a la hora de comunicarnos con las máquinas. Uno de los mayores y conocidos logros es Siri[9], la aplicación de reconocimiento del habla de la compañia Apple[10]. Actualmente existen nuevos competidores como Cortana (de Microsoft) o el Google Home[11] de Google.

- **Traducción automática:** la idea de traducir textos y encontrar un lenguaje universal para todos se remonta al siglo XVII. En la actualidad podemos encontrar entre los traductores más destacados el de Google así como la predicción de textos de los dispositivos móviles.
- **Síntesis de voz:** se trata de la producción artificial de habla. En este campo se puede destacar el sintetizador de voz del famoso científico Stephen Hawking.
- **Generación de lenguajes naturales:** es el proceso de construcción de términos, basados en un lenguaje natural, aplicables a expresiones o partes de ellas. Puede ser hablado o escrito. Un caso bastante extravagante es el de uno de los proyectos de Facebook. La compañía estadounidense estaba trabajando en un proyecto para mejorar los *chatbots* de la red social. Para ello, desarrolló dos *bots* que utilizaban el inglés como lenguaje natural, que trataban de mantener una conversación sobre negocios y que iban aprendiendo gracias a las redes neuronales. Con el tiempo, ambos *bots* consiguieron desarrollar una variación del lenguaje original mucho más lógico y simple que les permitía comunicarse sobre el tema que trataban[12]. En la figura 2.19 se puede apreciar un fragmento de la conversación que mantuvo esta inteligencia artificial.

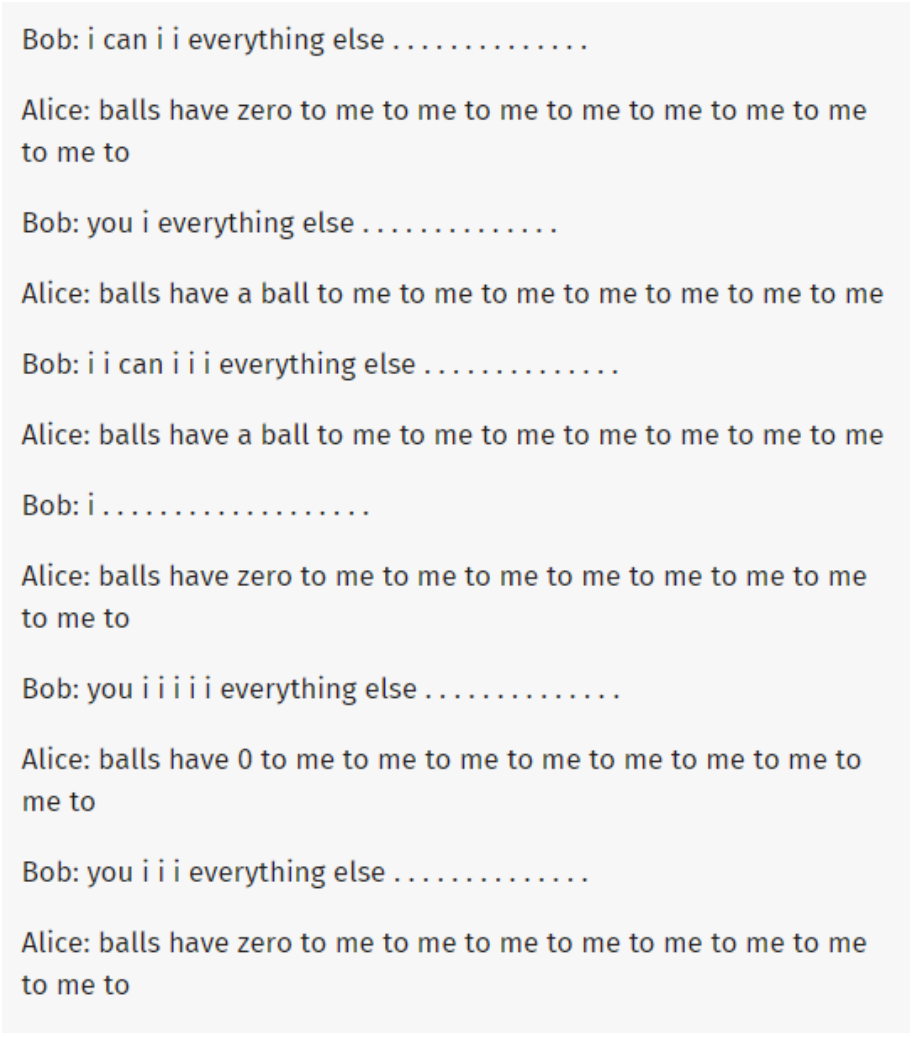

Figura 2.19: Conversación entre 2 *chatbot* de Facebook (Fuente: `http://www.independent.co.uk/`).

2.2. Agentes automáticos, Videojuegos y Simuladores

Desde su aparición en los años 50 del siglo pasado, los videojuegos han tenido un crecimiento y desarrollo espectaculares. Y los números son abrumadores. Según el estudio anual de la Asociación de Software de Entretenimiento (Entertainment Software Association, ESA), sólo en Estados Unidos el 67 % de las familias tiene un dispositivo en cuál se usa para jugar a videojuegos. El gran potencial de esta industria se puede observar en el dinero que mueve. En 2016, 30.400 millones de $ fueron gastados por los consumidores en la industria del videojuego. En la figura 2.20 se puede observar el crecimiento de esta cantidad. Tan sólo en el contenido (videojuegos físicos y digitales), los ciudadanos estadounidenses invirtieron más de 24 mil millones de $, sin tener en cuenta el *hardware* (consolas y pc's) ni los accesorios (como los elementos de realidad virtual)[13].

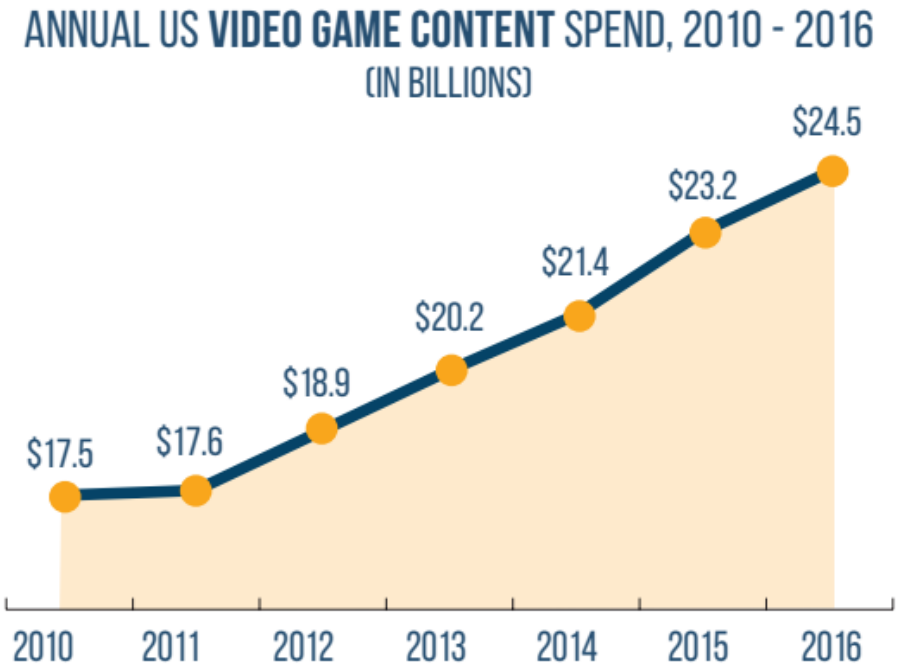

Figura 2.20: Miles de millones de dólares gastados en videojuegos en EEUU (2010-2016) (Fuente: `http://www.theesa.com`).

Estos datos cubren tan solo la parte de Estados Unidos. La figura 2.21 muestra un gráfico del total de dinero (en dólares) que generaron los videojuegos en el año 2016[14]. Es tan potente este mercado y el crecimiento del mismo, que se realizan estimaciones del dinero que pueda llegar a generar la industria del videojuego (tal cual se muestra en la figura 2.22[3]).

[3]En esta figura además se hace una división de los diferentes mercados/dispositivos que engloba la industria del entretenimiento.

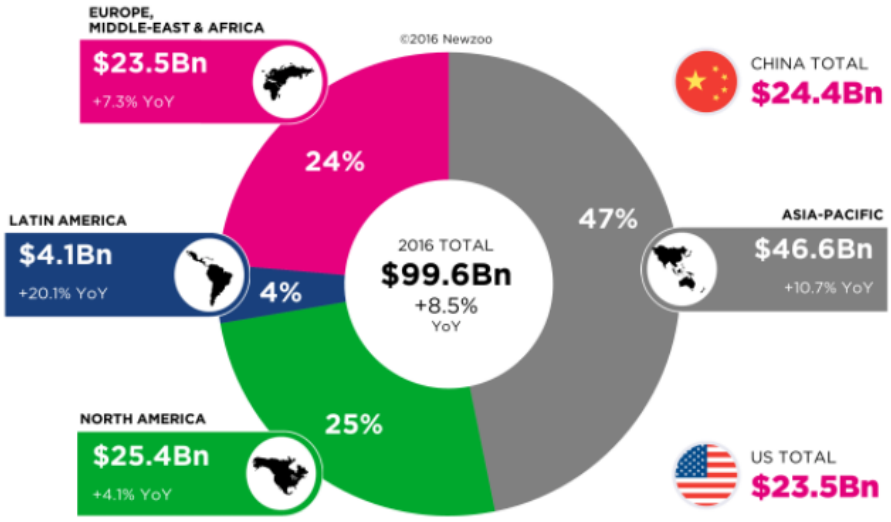

Figura 2.21: Miles de millones de dólares gastados en videojuegos en 2016 en todo el mundo (Fuente: `https://newzoo.com`).

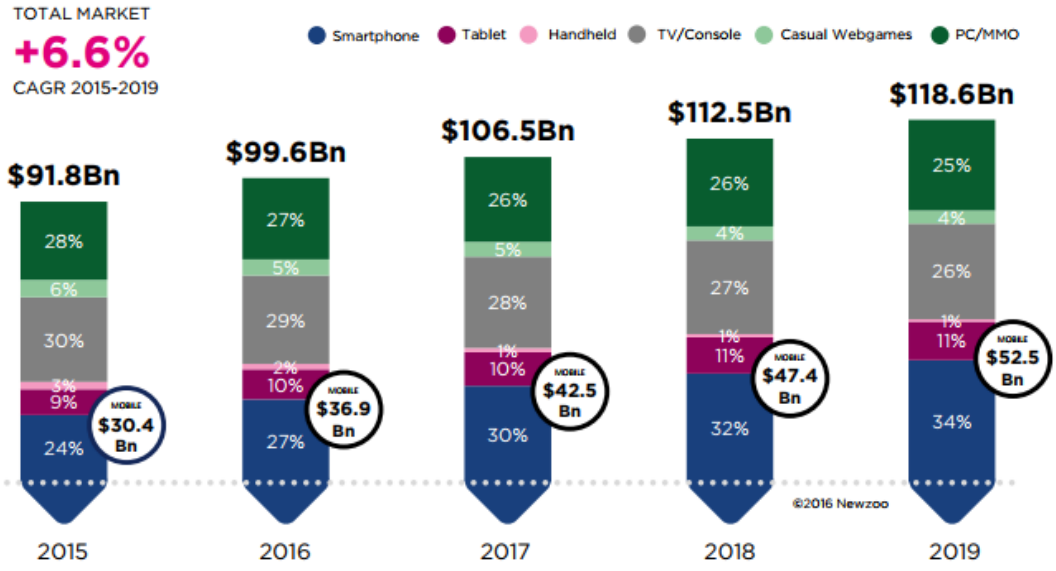

Figura 2.22: Estimación del dinero gastado en videojuegos entre 2015 y 2019 (Fuente: `https://newzoo.com`).

A parte del mercado norteamericano, el chino es otro de los más grandes en este sector. Sólo en los últimos 3 años, el mercado chino ha conseguido irrumpir en estas estadísticas de forma sensacional. Tan sólo en el año pasado, el 58 % del crecimiento de ingresos del mercado provino de la región de Asia y Pacífico, en la cual China tiene una incidencia de más del 50 %.

Pero, ¿de dónde nace todo esto? Como ya se ha dicho, hay que remontarse a la década de los 50. No queda muy claro cuál fue el primer videojuego de toda la historia, puesto que hay gente que asegura que fue el simulador de lanzamiento de misiles de Thomas T. Goldsmit y Estle Ray Mann, otros afirman que fue NIMROD (una

computadora diseñada exclusivamente para jugar el juego NIM), mientras que el resto manifiesta que fue OXO (versión de tres en raya para computadoras)[15]. Lo que sí está claro es que con la llegada de Pong y, poco más tarde, Spacewar, la industria se asentó en nuestro mundo. A los pocos años llegó la primera consola (los juegos anteriores estaban programados para ordenadores) y con ella los cartuchos de videojuegos. A partir de la década de los 70 aparecieron juegos de 1 jugador contra la máquina, la cuál manejaba enemigos que seguían ciertos patrones almacenados. Space Invaders fue uno de los primeros videojuegos en añadir algún tipo de inteligencia artificial a los enemigos del juego, los cuales eran capaces de aprender cierto movimientos del jugador y responder a ellos. Con la llegada de Pac-Man se introdujeron los algoritmos de búsqueda y se empezaron a usar las primeras máquinas de estados. Pero no fue hasta los años 90 cuando se produjo el alzamiento de las técnicas de IA en los videojuegos.

En la actualidad se utilizan numerosas y diversas técnicas e, incluso, se combinan unas con otras. Los próximos apartados se centran en los aspectos que implican este proyecto: máquinas de estados en los videojuegos y simuladores.

2.2.1. Videojuegos y Maquinas de estados

La introducción de las máquinas de estados en los videojuegos pasa por la aparición de los llamados agentes inteligentes. Un agente inteligente se puede definir como una entidad que percibe el entorno que le rodea (en este caso el juego) y es capaz de actuar de forma razonada sobre el mismo en base a lo que ha percibido[16]. En la figura 2.23 se puede observar un diagrama de un agente inteligente. Mediante los sensores percibe el estado del juego (del entorno), en base a unas condiciones y normas que tiene predefinidas escoge las acciones a realizar y, mediante los emisores, lleva a cabo las acciones escogidas las cuales repercuten en el entorno.

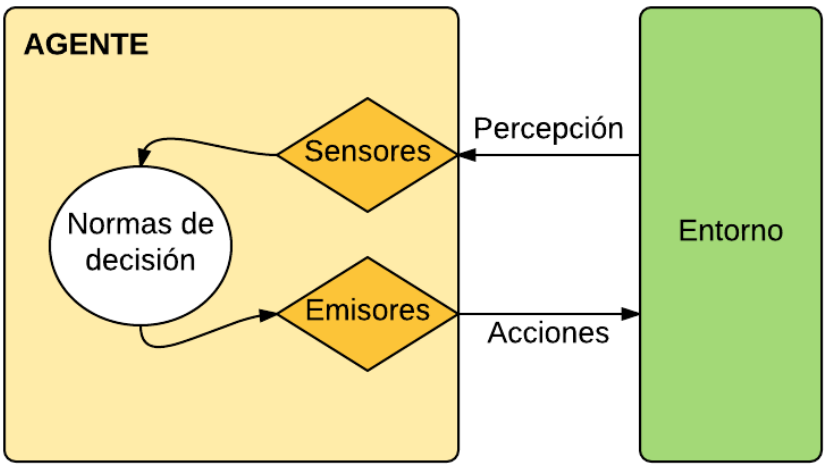

Figura 2.23: Diagrama de agente inteligente (Fuente: elaboración propia).

Los agentes inteligentes están dotados de autonomía y cierta inteligencia que puede ser modificada si se añaden técnicas de aprendizaje. De ahí su similitud con las máquinas de estados. Además, ambos pueden ser combinados para obtener sistemas más complejos y que aparenten tener una mayor inteligencia.

Uno de los primeros juegos que introdujo las maquinas de estados en la industria del videojuEgo fue el Pac-Man. En la figura 2.24 se puede observar un ejemplo de un autómata finito que controla uno de los fantasmas del juego. Una de las principales anécdotas de este juego es que que tiene 4 fantasmas diferentes y cada uno se caracteriza de una forma diferente, lo que hace que cada fantasma tenga un comportamiento distinto. Además, algunos de los fantasmas trabajan en grupo para conseguir atrapar a Pac-Man.

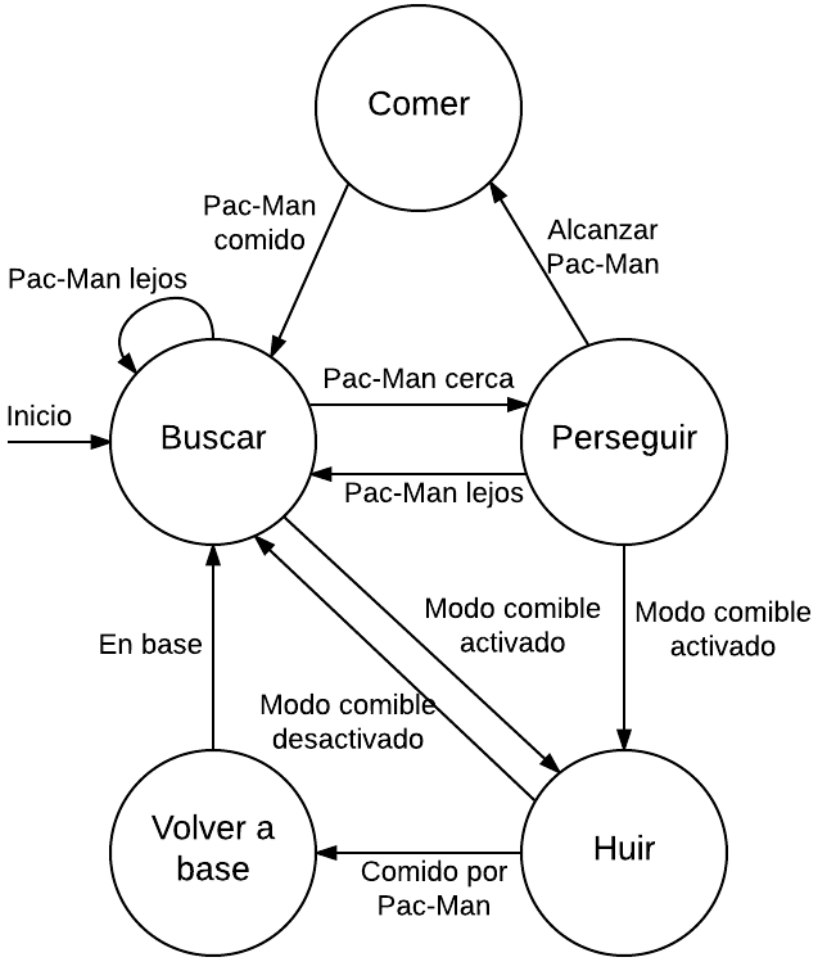

Figura 2.24: Ejemplo de un AFD de un fantasma del Pac-Man (Fuente: elaboración propia).

Como se ve en la figura 2.23, el interior de un agente inteligente es un conjunto de reglas, condiciones y normas, que pueden ser perfectamente definidas por un autómata finito. Pero, a su vez, también se pueden establecer a través de bucles y condicionales en código. Esto quiere decir que cualquier AF puede ser programado como un conjunto de condicionales y bucles, lo que hace que a veces no se vean tan presentes en otros trabajos.

Motores de videojuegos

Los motores de videojuegos han sido uno de los mejores inventos para poder desarrollar videojuegos. Gracias a ellos el modelado, la programación y la aplicación de

inteligencia artificial a los juegos se ha simplificado. Además, permiten una mejora de la calidad de éstos. A continuación se hablará sobre 2 de los motores de videojuegos más conocidos y utilizados en la industria actual, aunque existen muchos otros, cada uno con sus características.

- **Unity[17]:** es uno de los motores de videojuegos que más ha despuntado en los últimos años. Gracias a todas las funcionalidades que ofrece, así como los aportes que puede hacer la comunidad para mejorarlo, han hecho de esto motor uno de los mejores del mercado. Además, la iniciación al mismo es muy simple para un usuario que no tenga mucha experiencia en el sector y con este tipo de software. Unity permite añadir funcionalidad a sus personajes y comportamientos del juego mediante máquinas de estados. Utilizando la interfaz del sistema, de forma gráfica se pueden crear las MEF y definir cada estado y transición de la misma. En la figura 2.25 se puede observar un ejemplo de una máquina de estados creada con la interfaz de Unity.

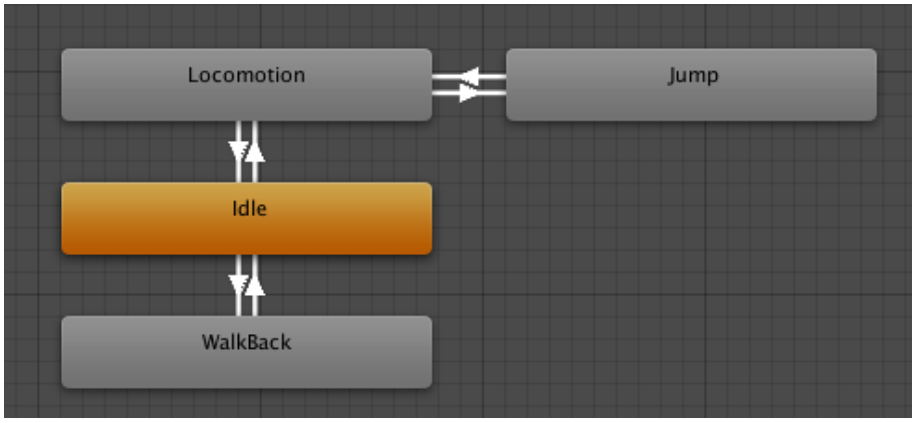

Figura 2.25: Ejemplo de una MEF creada en Unity (Fuente: `https://docs.unity3d.com/Manual`).

  Unity ofrece un gran abanico de oportunidades para los desarrolladores de videojuegos (sobretodo en el sector *indie*) y por ello es uno de los más escogidos por la comunidad. Algunos de los juegos más conocidos desarrollados con este motor son:

  - Rust (`www.playrust.com`)
  - Ori and the Blind Forest (`https://www.orithegame.com`)
  - Firewatch (`http://www.firewatchgame.com`)
  - Hearthstone: Heroes of Warcraft (`https://eu.battle.net/hearthstone/es`)

- **Unreal Engine[18]:** creado por la compañía Epic Games[19], es uno de los motores de videojuegos más antiguos que existen actualmente y, por lo tanto, es uno de los más desarrollados. Unreal se caracteriza por haber realizado un gran progreso en el diseño. Esto se puede observar en sus modelos realistas (favorecidos también por una iluminación bastante natural) como se ve en la figura 2.26. Comparado con Unity, Unreal permite desarrollar videojuegos más realistas, pero a su vez más complejos y costosos.

Figura 2.26: Diseño de 2 coches creados con la tecnología de UE (Fuente: `https://www.unrealengine.com`.

En cuanto a las máquinas de estados, al igual que Unity, UE posee una interfaz que te permite crear contenido de forma gráfica. Además, se puede complementar con código, lo que permite al desarrollador ver ambos enfoques de las máquinas de estados. En la figura 2.27 se puede observar un ejemplo de máquina de estados diseñada con la interfaz de Unreal.

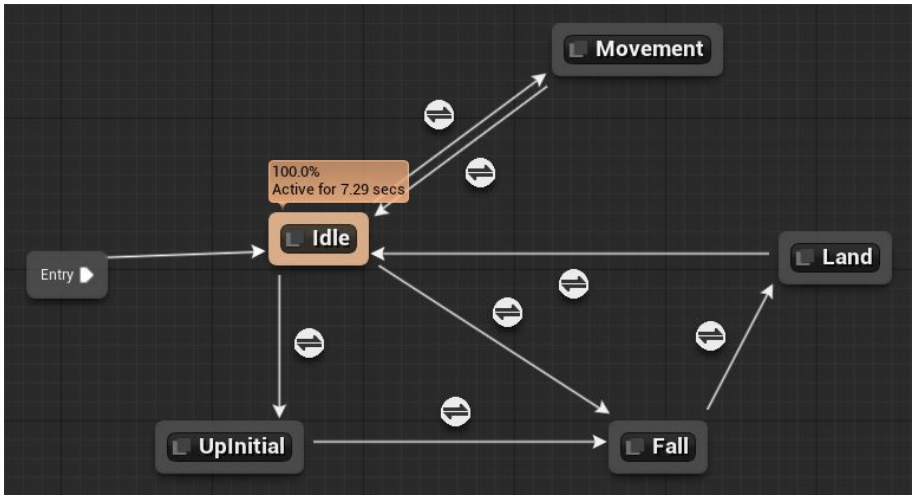

Figura 2.27: Ejemplo de una MEF creada en Unreal Engine (Fuente: `https://docs.unrealengine.com`.

Entre los juegos más conocidos creados con UE4 (última versión del motor) encontramos:

- Paragon (`https://www.epicgames.com/paragon`)
- Gears of War 4 (`https://gearsofwar.com`)
- Dragon Ball Fighter Z (`https://www.bandainamcoent.com/games/dragon-ball-fighterz`)
- Unreal Tournament (`https://www.epicgames.com/unrealtournament`)

Algunos de estos motores son tan potentes, que permiten crear contenido de realidad virtual tan realista que, al observarlo, parezca que te sumerjas en una segunda realidad. Además, algunas de las cinemáticas de películas de animación y cortometrajes, también se llegan a desarrollar con estas herramientas.

Ejemplo de máquina de estados en videojuegos

En las secciones anteriores se ha comentado el estado actual de las máquinas de estados así como de los videojuegos en general. Pero, a parte del Pac-Man, no se ha mencionado ningún juego que utilice las máquinas de estados en la inteligencia artificial del mismo. Como se ha visto anteriormente, las máquinas de estados pueden ser utilizadas para controlar numerosos procesos o situaciones. Y en los videojuegos ocurre lo mismo. Es una de las tecnologías más utilizas por los desarrolladores de videojuegos. Se pueden utilizar tanto para controlar el estado del juego, como por ejemplo ocurre con la saga *Uncharted*[20], en la cual se emplean, entre otras situaciones, para controlar la apertura de una puerta o la destrucción de objetos, como para manejar la ïnteligencia.º el comportamiento de los personajes y NPC's (Non-Player Character) del juego, como ocurre con los fantasmas de Pac-Man o con los personajes del Unreal Tournament.

Pero escapando de las altas esferas de esta industria, también se pueden encontrar ejemplos muy significativos. Entre estos me gustaría destacar los siguientes:

- **Evolutivo:** se trata de un videojuego de peleas desarrollado por Rafael Norman Saucedo Delgado[21]. En este juego, se utiliza máquinas de estados y búsqueda de caminos jerárquica. En el juego hay 2 personajes que se enfrentan: uno está controlado por el jugador y el otro (el rival) lo controla la IA. El objetivo es que el rival aprenda de tus movimientos y sea capaz de desarrollar una estrategia o combinación de movimientos y golpes para enfrentarse a ti y ganarte. Las máquinas de estados son utilizadas en este caso para conseguir determinar la estrategia que tiene que seguir el enemigo. Saucedo ha realizado una adaptación en código la cual se encuentra presente en GitHub. El trabajo inicial pertenece a M. Rimachi, L. Bardalez y D. Achanccaray y fue desarrollado para la Universidad Nacional de Ingeniería de Lima (Perú)[22].

- **Vanquish y Bayonetta:** son dos juegos diferentes, pero que en este caso están relacionados por el trabajo de Vanessa Giannaccini Sapsezian[23]. Esta graduada en Diseño de Videojuegos por la Universidad Anhembi Morumbi de São Paulo (Brasil) ha diseñado una máquina de estados para un personaje de cada uno de los 2 juegos. En ambos casos, se trata de uno de los "jefes" (boss) finales de alguno de los capítulos de la historia del videojuego. En las figuras 2.28 y 2.29 se puede observar la complejidad de ambas MEF.

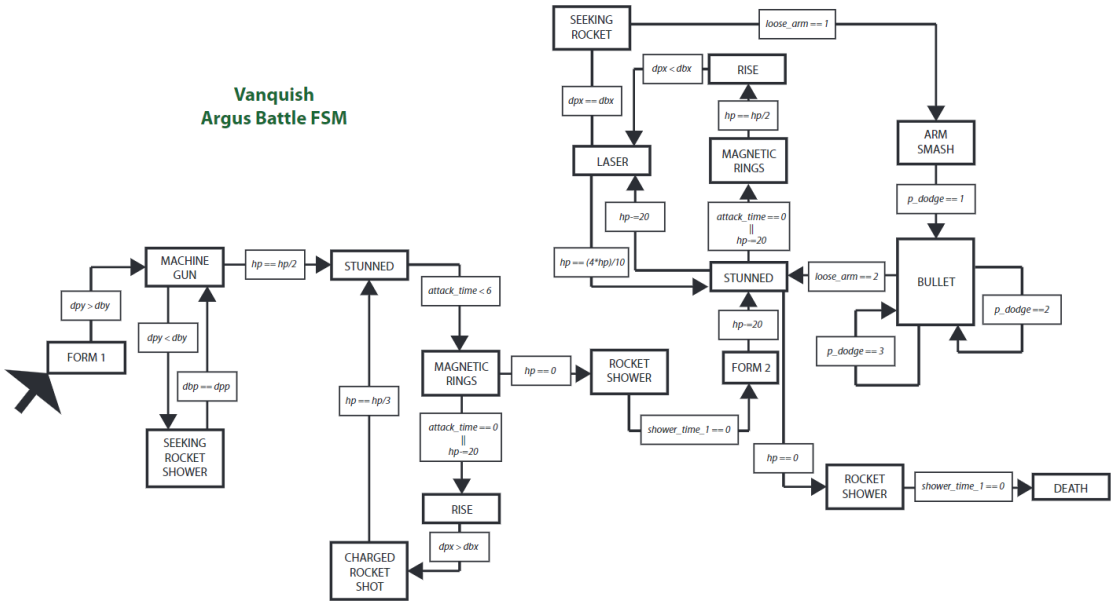

Figura 2.28: MEF para el *boss* Argus de Vanquish (Fuente: `https://munchyfly.me`).

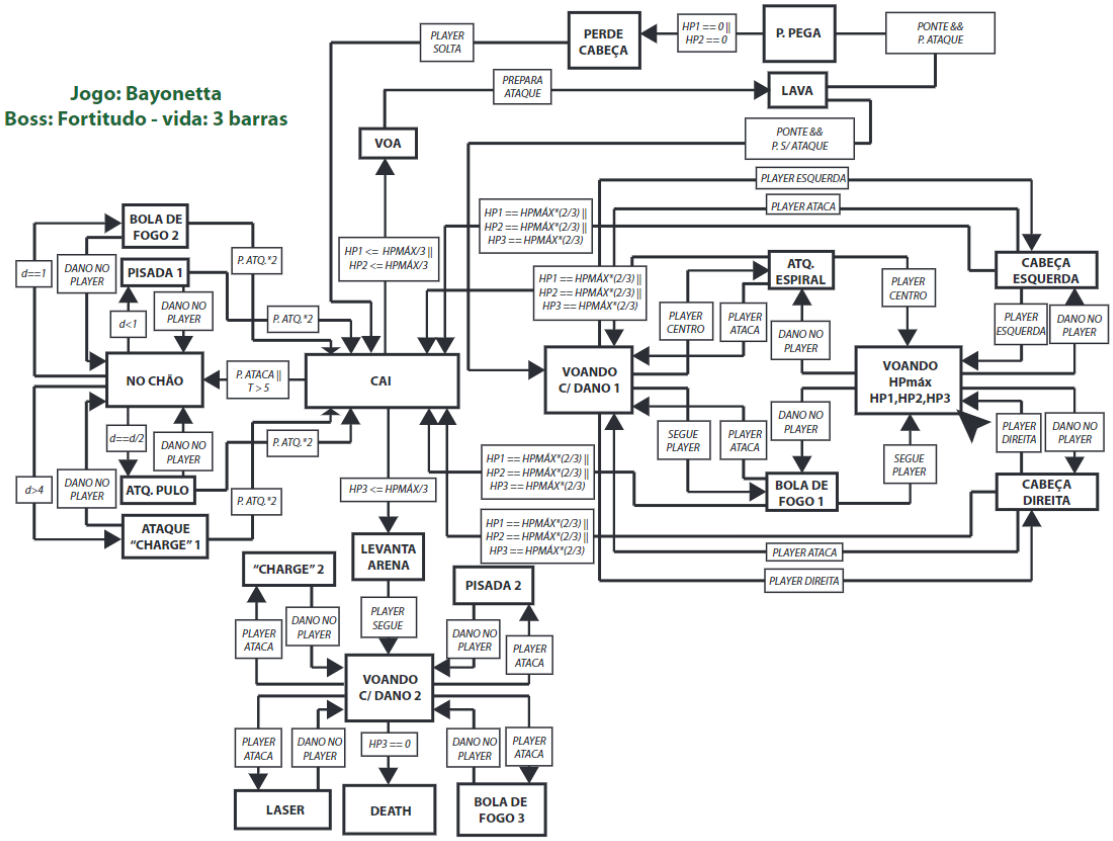

Figura 2.29: MEF para el *boss* Fortitudo de Bayonetta (Fuente: `https://munchyfly.me`).

2.2.2. Simuladores de conducción

Si bien The Open Racing Car Simulator se trata de un videojuego, también es un simulador de conducción. Por ello, creo que resulta conveniente hablar sobre el estado actual de los simuladores de conducción.

Existen tipos variados de simuladores: desde simuladores que se desarrollan para el entretenimiento (como pueden ser los videojuegos), hasta simuladores empleados para el aprendizaje y formación de conductores. En la simulación no solo se tiene en cuenta el software, sino que el hardware también es una parte muy importante de ésta, ya que puede llegar a permitir al usuario alcanzar el toque de realidad que se busca con los simuladores. Pero la parte que realmente importa a la IA es el software, y por ello se puede decir que TORCS es un simulador de conducción, ya que permite investigar y desarrollar conceptos y técnicas de la inteligencia artificial.

A continuación, se hará mención de alguno de los simuladores más importantes y conocidos que existen. La información que se muestra a continuación está obtenida del Trabajo de Fin de Grado de Victor Manuel Zamora España[24].

- **NADS:** *National Advanced Driving Simulator*, es el simulador de conducción más sofisticado del mundo. Está desarrollado por la *National Highway Traffic Safety Administration* (NHTSA). Dispone de un hardware y software muy avanzado y que permite al conductor experimentar sensaciones muy similares a las que experimentaría en un entorno real de conducción. Se utiliza para la investigación de diferentes situaciones de conducción, manteniendo al piloto seguro durante toda la simulación.

- **Forza Motorsport:** es una de las sagas de videojuegos más realistas que existe a nivel automovilístico. Cada vehículo es diseñado intentando respetar al máximo las características reales del mismo. Además, permite ser jugado con un sistema de conducción instalado, lo que le otorga una mayor sensación de realidad.

- **The Most Realistic Racing Simulator:** se trata del simulador de carreras más realista del mundo. Los simuladores de Fórmula 1 son una variante adaptada de éste. Está equipado con servo actuadores lineales que simulan la suspensión del coche y con un monocasco capaz de rotar 306° con una aceleración de hasta 0'5G.

2.3. TORCS

The Open Racing Car Simulator o TORCS[25], es un simulador de carreras de coches multiplataforma. Una de sus principales aplicaciones es como juego de carreras, pero también juega un papel importante como plataforma de investigación y desarrollo en la inteligencia artificial. Esto es debido a que se trata de un juego de código abierto, es decir, que cualquier persona de la comunidad puede realizar aportes al mismo para mejorarlo. Además, la gran variedad tanto de coches como te trazados, así como de oponentes contra los que competir y de la aproximación a la realidad en la parte de componentes y sensores de los coches, hacen de este simulador una herramienta de lo más adecuada para el desarrollo de la IA.

Sí es verdad que en la parte gráfica el videojuego pierde mucha calidad (como se puede observar en la figura 2.30), pero como lo que se busca es poder investigar los diferentes aspectos de la IA en los que TORCS pueda ayudar, los gráficos son la parte menos importante. Incluso, esto no deja de ser un beneficio, ya que nos permite ejecutar el simulador en ordenadores menos potentes, es decir, a un menor coste.

Figura 2.30: Captura del TORCS durante una carrera (Fuente: `https://sourceforge.net/projects/torcs`).

Historia

TORCS fue creado por Eric Espiè y Christophe Guionneau en 2004, pero al tratarse de un proyecto de código libre, las aportaciones de diferentes contribuidores han conseguido hacer que el juego crezca. Actualmente, el proyecto se encuentra dirigido por Bernhard Wymann[26], quién ha realizado grandes aportes a la comunidad. En el apartado de `Créditos` de la página oficial se pueden observar a todos los contribuidores (no anónimos) que han colaborado en el desarrollo de este videojuego.

Versiones

Desde de que se creó, el juego ha ido pasando por diferentes versiones recibiendo mejoras tanto de sus propios programadores como de la comunidad. Actualmente, en la página oficial del proyecto[27] se encuentran disponibles algunas de estas versiones (las más antiguas ya no están):

- `1.2.3`: es la versión más antigua disponible. Dado que se trata de una versión del año 2005 no se le da uso al estar demasiado desactualizada.
- `1.2.4`
- `1.3.0`
- `1.3.1`: con la actualización a la versión `1.3` el juego mejoró en el aspecto del desarrollo, y es a partir de esta revisión cuando se ven más usos del simulador.

- `1.3.2`
- `1.3.3`
- `1.3.4`: fué en la versión `1.3.3` cuando llegó una actualización importante al juego que permitía realizar entrenamientos de la IA mediante la consola de comandos. Pero la versión que realmente ha triunfado ha sido esta gracias a las pequeñas mejoras que la comunidad aportó a la versión anterior.
- `1.3.5`
- `1.3.6`
- `1.3.7`: se trata de la versión más reciente (2016) y está empezando a ser la más utilizada gracias a la nueva documentación que trae con ella.

Competiciones

Una manera útil y a la vez divertida de probar los desarrollos que hace la comunidad en el juego son las competiciones. Desde que se sacaron las primeras versiones estables (2005) y que el público empezó a utilizar para desarrollar sus propios *bots*, se han realizado competiciones oficiales y no oficiales utilizando el TORCS. Es así como se puede encontrar en la página oficial de Bernhar Wymann[26] un apartado sobre las competiciones oficiales que se realizan (*TORCS Racing Board, TRB*). En esta plataforma se puede encontrar diferente información a cerca del juego, de los participantes, coches, etc., además del apartado principal de eventos. Se han realizado competiciones todos los años desde el 2005, exceptuando 2009 y 2010 (no se explica el por qué).

También existen otras páginas que han organizado competiciones de simulación de carreras en las cuales se ha utilizado del TORCS como videojuego. Es el caso del Campeonato de Carreras Simuladas de Coches (en inglés Simulated Car Racing Championship, SCRC)[28]. Se trata de un campeonato organizado por 4 investigadores de la Universidad de Adelaida[29] y la Universidad Politécnica de Milán[30] en conjunto. Llevan organizando este tipo de campeonatos desde 2007, exceptuando 2008 y 2014, y utilizando el TORCS como plataforma de simulación desde el año 2012. Al igual que en la página oficial del juego, en este sitio web pueden encontrarse diferentes agentes desarrollados por los competidores de los campeonatos. Este *software* está desarrollado en diferentes lenguajes como C++, Java o Python y han servido de modelo y guía para la realización de este proyecto. Además, en la página web del SCRC[28] se puede encontrar un manual el cuál se ha utilizado como guía para llevar a cabo los primeros pasos de descarga e instalación del *software* necesario. En el Anexo de este documento se ha realizado una adaptación de dicho manual con los pasos seguidos para este trabajo.

Capítulo 3

Análisis del sistema

Como se ha mencionado en la sección 1.2, este proyecto está orientado a diseñar y crear el control de un agente automático de un videojuego. En nuestro caso sólo es necesario centrarse en el desarrollo del cliente, puesto que tanto la plataforma como el servidor ya están implementados. Por consiguiente, en las siguientes secciones se especificarán los casos de uso y los requisitos necesarios para desarrollar este sistema.

3.1. Casos de uso

Un caso de uso es la descripción de los pasos o actividades que debe seguir un actor (usuario del sistema) para llevar a cabo un proceso concreto en el sistema. Los casos de uso pueden definirse en forma de tabla o en forma de diagrama.

3.1.1. Descripción tabular

Las tablas que describen cada caso de uso deben seguir un formato específico que se define a continuación en la tabla 3.1:

Tabla 3.1: Plantilla para los casos de uso

Identificador	CU-XX
Título	
Objetivo	
Precondiciones	
Postcondiciones	
Escenario de éxito	

A continuación se explicará el significado de cada una de las entradas de la tabla:

- **Identificador:** es la representación unívoca del caso de uso. Los caracteres que lo identifican son:
  - **CU:** caso de uso.
  - **XX:** número de caso de uso.
- **Título:** nombre del caso de uso.

- **Objetivo:** situación que se debe alcanzar con el caso de uso.
- **Precondiciones:** condiciones que han de cumplirse antes de realizar el caso de uso.
- **Postcondiciones:** condiciones que ha de cumplirse después de realizar el caso de uso.
- **Escenario de éxito:** procedimiento a seguir para llevar a caso el caso de uso.

De la tabla 3.2 a la tabla 3.10 se especifican todos los casos de uso que cubre este proyecto.

Tabla 3.2: CU-01

Identificador	CU-01
Título	Acelerar
Objetivo	El coche aumenta su velocidad.
Precondiciones	▪ Estar dentro de una carrera. ▪ Tener el cliente conectado al servidor. ▪ Velocidad baja.
Postcondiciones	▪ Pedal de acelerador pisado. ▪ La velocidad del coche irá aumentando.
Escenario de éxito	▪ El sistema detecta por los sensores que se encuentra en una situación para acelerar. ▪ La máquina de estados transita al estado de aceleración. ▪ El agente envía la acción de acelerar al juego. ▪ El coche acelera.

Tabla 3.3: CU-02

Identificador	CU-02
Título	Frenar
Objetivo	El coche reduce su velocidad.
Precondiciones	▪ Estar dentro de una carrera. ▪ Tener el cliente conectado al servidor. ▪ Volocidad elevada.
Postcondiciones	▪ Pedal de freno pisado. ▪ La velocidad del coche irá disminuyendo.
Escenario de éxito	▪ El sistema detecta por los sensores que se encuentra en una situación para frenar. ▪ La máquina de estados transita al estado de freno. ▪ El agente envía la acción de frenar al juego. ▪ El coche decelera.

Tabla 3.4: CU-03

Identificador	CU-03
Título	Control del embrague
Objetivo	El sistema simula que el piloto pisa el pedal de embrague.
Precondiciones	▪ Estar dentro de una carrera. ▪ Tener el cliente conectado al servidor. ▪ Tener las RPM muy altas.
Postcondiciones	▪ Pedal de embregue pisado.
Escenario de éxito	▪ El sistema detecta por los sensores que se encuentra en una situación para cambiar de marcha. ▪ La máquina de estados transita al estado de pisar embrague. ▪ El agente envía la acción de embragar al juego. ▪ El coche queda embragado.

Tabla 3.5: CU-04

Identificador	CU-04
Título	Subir marcha
Objetivo	El coche sube una marcha.
Precondiciones	▪ Estar dentro de una carrera. ▪ Tener el cliente conectado al servidor. ▪ La marcha actual es inferior a la máxima marcha. ▪ El embrague está pisado. ▪ Las RPM son elevadas.
Postcondiciones	▪ La marcha es un valor superior al anterior. ▪ Las RPM son bajas.
Escenario de éxito	▪ El sistema detecta por los sensores que se encuentra en una situación para subir de marcha. ▪ La máquina de estados transita al estado de subir marcha. ▪ El agente envía la acción de subir marcha al juego. ▪ El coche sube de marcha.

Tabla 3.6: CU-05

Identificador	CU-05
Título	Bajar marcha
Objetivo	El coche baja una marcha.
Precondiciones	▪ Estar dentro de una carrera. ▪ Tener el cliente conectado al servidor. ▪ La marcha actual es superior a la mínima marcha. ▪ El embrague está pisado. ▪ Las RPM son bajas.
Postcondiciones	▪ La marcha es un valor inferior al anterior. ▪ Las RPM son altas.
Escenario de éxito	▪ El sistema detecta por los sensores que se encuentra en una situación para bajar de marcha. ▪ La máquina de estados transita al estado de bajar marcha. ▪ El agente envía la acción de bajar marcha al juego. ▪ El coche baja de marcha.

Tabla 3.7: CU-06

Identificador	CU-06
Título	Girar izquierda
Objetivo	El coche gira a la izquierda.
Precondiciones	▪ Estar dentro de una carrera. ▪ Tener el cliente conectado al servidor. ▪ El eje del circuito está orientado hacia la izquierda del coche.
Postcondiciones	▪ El coche queda en línea con el eje del circuito.
Escenario de éxito	▪ El sistema detecta por los sensores que se encuentra en una situación para girar ala izquierda. ▪ La máquina de estados transita al estado de girar a la izquierda. ▪ El agente envía la acción de girar a la izquierda al juego. ▪ El coche gira a la izquierda.

Tabla 3.8: CU-07

Identificador	CU-07
Título	Girar derecha
Objetivo	El coche gira a la derecha.
Precondiciones	▪ Estar dentro de una carrera. ▪ Tener el cliente conectado al servidor. ▪ El eje del circuito está orientado hacia la derecha del coche.
Postcondiciones	▪ El coche queda en línea con el eje del circuito.
Escenario de éxito	▪ El sistema detecta por los sensores que se encuentra en una situación para girar ala derecha. ▪ La máquina de estados transita al estado de girar a la derecha. ▪ El agente envía la acción de girar a la derecha al juego. ▪ El coche gira a la derecha.

Tabla 3.9: CU-08

Identificador	CU-08
Título	Enviar información
Objetivo	El agente envía información al juego/servidor.
Precondiciones	▪ El juego/servidor están en ejecución. ▪ El agente está en ejecución.
Postcondiciones	▪ El servidor ha recibido información del agente.
Escenario de éxito	▪ Se lanza el juego/servidor. ▪ Se lanza el agente. ▪ El agente envía información al servidor para conectar con él.

Tabla 3.10: CU-09

Identificador	CU-09
Título	Recibir información
Objetivo	El agente recibe información del juego/servidor.
Precondiciones	▪ El juego/servidor están en ejecución. ▪ El agente está en ejecución.
Postcondiciones	▪ El agente ha recibido una respuesta del juego/servidor.
Escenario de éxito	▪ Se lanza el juego/servidor. ▪ Se lanza el agente. ▪ El agente envía información al servidor para conectar con él. ▪ El agente recibe los datos del servidor.

3.1.2. Descripción gráfica

En este apartado se muestra mediante una representación gráfica los casos de uso vistos anteriormente. En la figura 3.1 se muestra el diagrama de casos de uso que se aplican a la máquina de estados que controla el agente, mientras que en la figura 3.2 se observan los casos de uso que afectan directamente al funcionamiento y comunicación del agente con el juego.

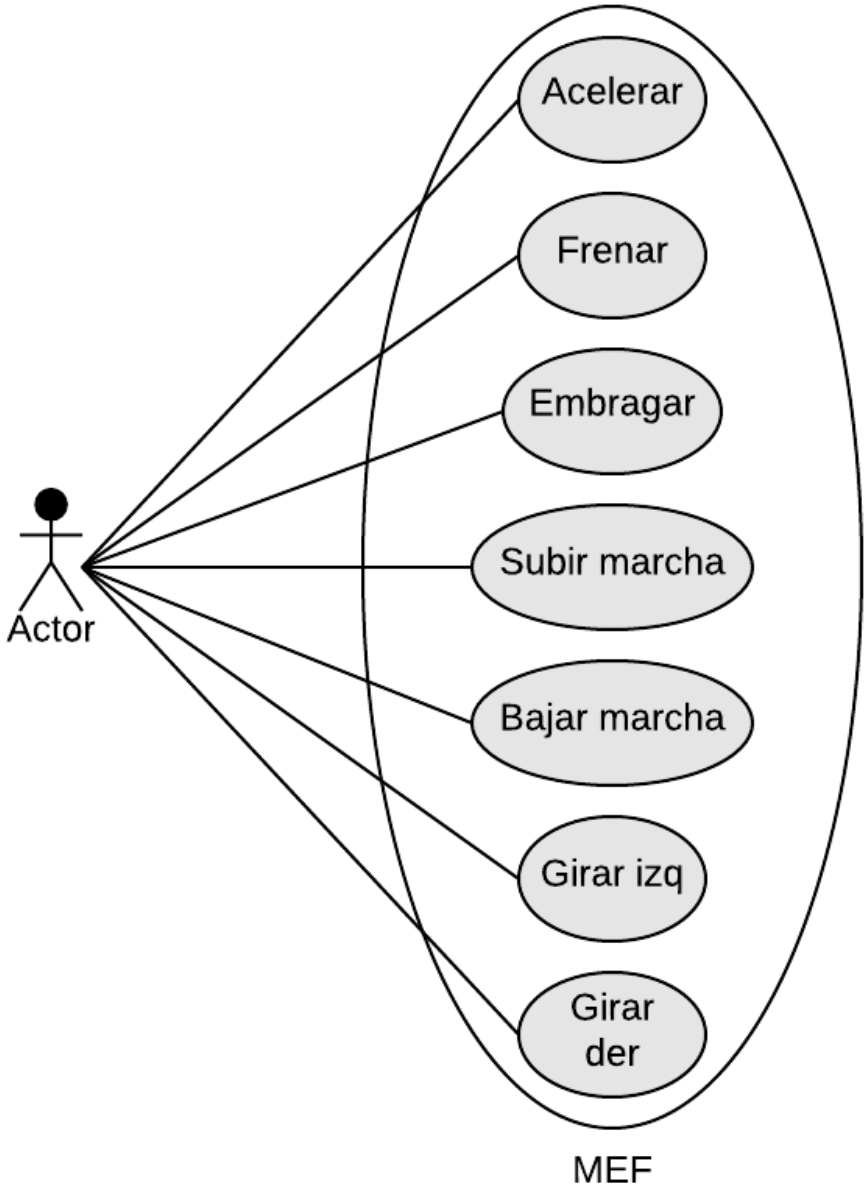

Figura 3.1: Diagrama de casos de uso de la MEF (Fuente: elaboración propia).

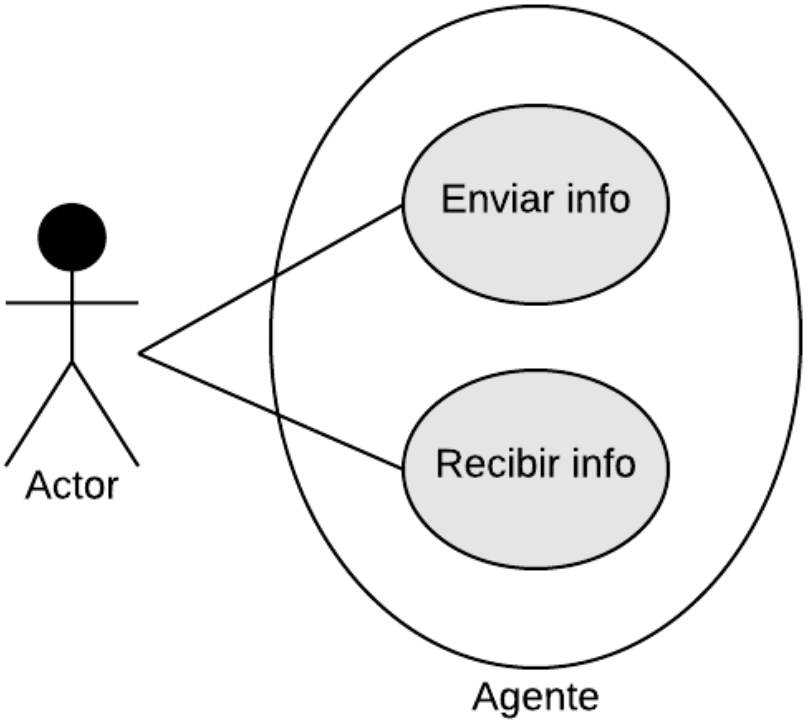

Figura 3.2: Diagrama de casos de uso del agente (Fuente: elaboración propia).

3.2. Especificación de requisitos

En esta sección se procede a describir los requisitos que debe cumplir el sistema desarrollado. En la tabla 3.11 se muestra el formato que seguirá cada requisito.

Tabla 3.11: Plantilla para los requisitos

Identificador	RT-XX		
Prioridad	☐ Alta	☐ Media	☐ Baja
Necesidad	☐ Esencial	☐ Recomendable	☐ Opcional
Verificabilidad	☐ Alta	☐ Media	☐ Baja
Estabilidad	☐ Alta	☐ Media	☐ Baja
Descripción			

A continuación se explicará el significado de cada una de los campos de la tabla:

- **Identificador:** es la representación unívoca del requisito. Los caracteres que lo identifican son:
  - **R:** requisito.
  - **T:** tipo de requisito (funcional - F; no funcional - NF).
  - **XX:** número de requisito.

- **Prioridad:** indica la prioridad con la que se debe cumplir el requisito. Puede ser: alta, media o baja.
- **Necesidad:** indica el grado de importancia del requisito para el desarrollo del proyecto. Puede ser: esencial, recomendable u opcional.
- **Verificabilidad:** grado en el cual se indica si la incorporación del requisito se puede evidenciar en el producto final. Puede ser: alta, media o baja.
- **Estabilidad:** representa el grado en el cual el requisito puede ser modificado durante el desarrollo del proyecto. Puede ser: alta, media o baja.
- **Descripción:** funcionalidad o restricción que implica el requisito en el proyecto..

Los requisitos de este proyecto se agrupan en dos tipos: funcionales y no funcionales. En los siguientes apartados se describen todos los requisitos del proyecto.

3.2.1. Requisitos funcionales

En este apartado se definen los requisitos funcionales del sistema. Este tipo de requisitos engloba todo modelo de comportamiento que el sistema debe cumplir: cálculos, detalles técnicos, manipulación de datos, etc. De la tabla 3.12 a la tabla 3.21 se muestran los requisitos funcionales del sistema.

Tabla 3.12: RF-01

Identificador	RF-01		
Prioridad	☒ Alta	☐ Media	☐ Baja
Necesidad	☒ Esencial	☐ Recomendable	☐ Opcional
Verificabilidad	☒ Alta	☐ Media	☐ Baja
Estabilidad	☒ Alta	☐ Media	☐ Baja
Descripción	El sistema será capaz de acelerar el coche durante una carrera.		

Tabla 3.13: RF-02

Identificador	RF-02		
Prioridad	☒ Alta	☐ Media	☐ Baja
Necesidad	☒ Esencial	☐ Recomendable	☐ Opcional
Verificabilidad	☒ Alta	☐ Media	☐ Baja
Estabilidad	☒ Alta	☐ Media	☐ Baja
Descripción	El sistema será capaz de frenar el coche durante una carrera.		

Tabla 3.14: RF-03

Identificador	RF-03		
Prioridad	☒ Alta	☐ Media	☐ Baja
Necesidad	☒ Esencial	☐ Recomendable	☐ Opcional
Verificabilidad	☒ Alta	☐ Media	☐ Baja
Estabilidad	☒ Alta	☐ Media	☐ Baja
Descripción	El sistema será capaz de subir de marcha durante una carrera.		

Tabla 3.15: RF-04

Identificador	RF-04		
Prioridad	☒ Alta	☐ Media	☐ Baja
Necesidad	☒ Esencial	☐ Recomendable	☐ Opcional
Verificabilidad	☒ Alta	☐ Media	☐ Baja
Estabilidad	☒ Alta	☐ Media	☐ Baja
Descripción	El sistema será capaz de bajar de marcha durante una carrera.		

Tabla 3.16: RF-05

Identificador	RF-05		
Prioridad	☒ Alta	☐ Media	☐ Baja
Necesidad	☒ Esencial	☐ Recomendable	☐ Opcional
Verificabilidad	☒ Alta	☐ Media	☐ Baja
Estabilidad	☒ Alta	☐ Media	☐ Baja
Descripción	El sistema será capaz de girar el coche a la izquierda durante una carrera.		

Tabla 3.17: RF-06

Identificador	RF-06		
Prioridad	☒ Alta	☐ Media	☐ Baja
Necesidad	☒ Esencial	☐ Recomendable	☐ Opcional
Verificabilidad	☒ Alta	☐ Media	☐ Baja
Estabilidad	☒ Alta	☐ Media	☐ Baja
Descripción	El sistema será capaz de girar el coche a la derecha durante una carrera.		

Tabla 3.18: RF-07

Identificador	RF-07		
Prioridad	☒ Alta	☐ Media	☐ Baja
Necesidad	☒ Esencial	☐ Recomendable	☐ Opcional
Verificabilidad	☒ Alta	☐ Media	☐ Baja
Estabilidad	☐ Alta	☒ Media	☐ Baja
Descripción	El sistema se comunicará con el servidor enviando y recibiendo información del mismo.		

Tabla 3.19: RF-08

Identificador	RF-08		
Prioridad	☒ Alta	☐ Media	☐ Baja
Necesidad	☒ Esencial	☐ Recomendable	☐ Opcional
Verificabilidad	☒ Alta	☐ Media	☐ Baja
Estabilidad	☒ Alta	☐ Media	☐ Baja
Descripción	El sistema será capaz de embragar durante una carrera.		

Tabla 3.20: RF-09

Identificador	RF-09		
Prioridad	☒ Alta	☐ Media	☐ Baja
Necesidad	☒ Esencial	☐ Recomendable	☐ Opcional
Verificabilidad	☒ Alta	☐ Media	☐ Baja
Estabilidad	☒ Alta	☐ Media	☐ Baja
Descripción	El sistema comunicará la/s máquina/s de estados con el controlador del agente.		

Tabla 3.21: RF-10

Identificador	RF-10		
Prioridad	☒ Alta	☐ Media	☐ Baja
Necesidad	☒ Esencial	☐ Recomendable	☐ Opcional
Verificabilidad	☒ Alta	☐ Media	☐ Baja
Estabilidad	☒ Alta	☐ Media	☐ Baja
Descripción	El sistema coordinará las máquinas de estados finitos que controlen la lógica del agente.		

3.2.2. Requisitos no funcionales

En este apartado de describen los requisitos no funcionales que debe seguir el sistema. Este tipo de requisitos hace referencia a las características de funcionamiento del sistema. Entre las tablas 3.22 y 3.24 se definen estos requisitos.

Tabla 3.22: RNF-01

Identificador	RNF-01		
Prioridad	☐ Alta	☒ Media	☐ Baja
Necesidad	☐ Esencial	☒ Recomendable	☐ Opcional
Verificabilidad	☒ Alta	☐ Media	☐ Baja
Estabilidad	☐ Alta	☒ Media	☐ Baja
Descripción	Los valores de las variables del agente se definirán en el sistema métrico.		

Tabla 3.23: RNF-02

Identificador	RNF-02		
Prioridad	☐ Alta	☒ Media	☐ Baja
Necesidad	☐ Esencial	☒ Recomendable	☐ Opcional
Verificabilidad	☒ Alta	☐ Media	☐ Baja
Estabilidad	☐ Alta	☒ Media	☐ Baja
Descripción	El sistema estará programado en Java.		

Tabla 3.24: RNF-03

Identificador	RNF-03		
Prioridad	☒ Alta	☐ Media	☐ Baja
Necesidad	☒ Esencial	☐ Recomendable	☐ Opcional
Verificabilidad	☐ Alta	☒ Media	☐ Baja
Estabilidad	☐ Alta	☒ Media	☐ Baja
Descripción	Se entenderá como usuario del sistema al agente que se haya programado.		

Capítulo 4

Arquitectura y diseño del sistema

En este capítulo del documento se presenta la arquitectura del sistema así como el diseño que se va a desarrollar. De manera más precisa, la meta de este punto del trabajo es definir cada componente que forma el sistema, el entorno en el que está englobado, las posibles interfaces que se utilice y las decisiones de diseño que se han tomado.

4.1. Visión general del sistema

El sistema que se va a desarrollar trata de un agente automático para un videojuego controlado por máquinas de estados finitos. Esto quiere decir que el sistema se ejecutará sobre un entorno simulado en el cual el diseño que se realice deberá cubrir una serie de tareas como:

- conectar con el juego,
- controlar el agente automático mediante una toma de decisiones,
- funcionar de una manera ”inteligente”.

Dado que se tratará de un agente diseñado específicamente para un videojuego, la portabilidad del mismo será prácticamente nula, al igual que ocurre con la MEF que controle su lógica. Lo que sí permitirá el diseño, será utilizar dicha máquina en distintos agentes del juego, es decir, controlar diferentes vehículos con variaciones en sus actores para observar cómo se comporta cada uno de ellos.

4.2. Estudio de la solución final

Teniendo en cuenta la especificación de requisitos y los casos de uso descritos en el capítulo 3, la solución que se escoja debe ser capaz de controlar diferentes aspectos del agente, pero sin limitar la manera en que se controlen. Esto quiere decir que existen un gran abanico de posibilidades tanto en la/s máquina/s de estados como en los medios que se utilicen (lenguaje, entorno de programación, estudios, etc.)[1].

[1]Aunque es cierto que esta afirmación entra en conflicto con el requisito no funcional 3.23, cabe destacar que este requisito ha sido modificado una vez elegida la solución que se iba a aplicar.

En primer lugar, se ha estudiado qué lenguaje era mejor para el desarrollo del sistema. Como se menciona en la sección 1.3, los lenguajes que se plantearon como posible lenguaje de programación han sido:

- C
- Matlab
- Python
- Java

C

Fue el primer lenguaje en el que se pensó por el bajo nivel que ofrece, pero la mayoría de las versiones de clientes estaban realizados en C++ antes que C. Por ello, y debido que los conocimientos de C++ no son los suficientes para poder llevar a cabo un proyecto como este, este lenguaje ha sido desestimado.

Matlab

Este lenguaje de programación fue propuesto por el tutor debido a la escasez de agentes que estaban desarrollados con él y porque aportaba un enfoque interesante sobre todo para el cálculo de los valores recibidos por los sensores del vehículo. Además, el entorno de desarrollo ofrece una herramienta de diseño (Simulink) con la cual se pueden definir máquinas de estados de forma gráfica y ver su funcionamiento mientras se ejecuta el sistema. Se ha probado a acoplar al agente la poca información y código que se ha encontrado e intentar definir una MEF mediante Simulink y enlazarla con el cliente. Pero los problemas que han surgido durante este proceso como *bugs* con el entorno de desarrollo y características específicas de la herramienta Simulink, han llevado al rechazo de este lenguaje.

Python

Se ha propuesto este lenguaje como una manera alternativa a los agentes que se han implementado otro años. El principal motivo de esta propuesta era la novedad del código en cuanto a la creación de un agente (se han encontrado pocos ejemplares desarrollados exclusivamente en Python). Se ha rechazado utilizar este lenguaje de programación mediante un mutuo acuerdo entre el tutor y el alumno porque no terminaba de convencer todo el proceso de desarrollo.

Java[2]

Ha sido la propuesta presentada después de descartar de inmediato el lenguaje C, ya que se trata del lenguaje sobre el que más información hay en el entorno en el que se va a trabajar (teniendo siempre en cuenta el videojuego y la tecnología que se quiere utilizar). A petición del alumno, el tutor y él han llegado a la conclusión de que se trata de la propuesta más viable debido al conocimiento del lenguaje y las bases en las

[2]Tras la elección de esta propuesta, se ha definido el requisito 3.23.

cuales se pueden apoyar para el desarrollo de todo el sistema.

El otro dilema a tratar, es la decisión de diseño de la máquina de estados finitos que se definirá para el sistema. Esta decisión de diseño se ha realizado describiendo diferentes posibles MEF's sin llegar a implementarlas para no sobrecargar la carga de trabajo, escribiendo en código únicamente la que se considere como solución del sistema. Para ello se tendrá en cuenta el número de máquinas de estados que se necesiten en cada propuesta así como la complejidad de cada una. En los apartado siguientes se explicarán algunos de los modelos más importantes que se han diseñado así como el diseño y arquitectura del modelo final.

4.3. Arquitectura del sistema

Antes de escoger un diseño para la MEF, es importante definir la arquitectura en la que se organiza el sistema. En esta sección del documento se van a presentar los módulos por los cuales el sistema está compuesto, así como se va a describir cada uno de ellos de la forma más detallada posible.

En este proyecto se pueden destacar dos módulos principales: el juego (o servidor) y el agente (o cliente). En la figura 4.1 se puede observar la arquitectura del sistema formada por estos 2 módulos.

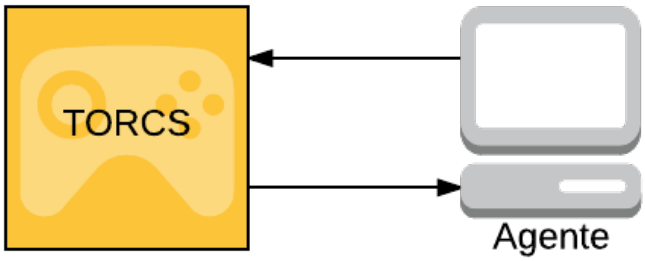

Figura 4.1: Arquitectura del sistema (Fuente: elaboración propia)

4.3.1. Juego

Podría calificarse como el módulo principal, puesto que comprende todo el entorno sobre el cuál el agente se va a ejecutar. El juego está integrado por dos submódulos:

- **Datos del juego.** En los datos del juego se encuentran todos los componentes que permiten la correcta ejecución del mismo: coches, circuitos, archivos de ejecución compilados, etc. No es necesario conocer estos datos en profundidad para poder desarrollar este sistema, basta con saber que están y que, si se desea experimentar una mayor experiencia del juego, se pueden modificar. Los datos que no está compilados, se encuentran en lenguaje XML.
- **Servidor.** Es la parte que nos permite enlazar el juego con los agentes que se desarrollan, y es por lo tanto por lo que al módulo del juego también se le puede denominar como servidor. El servidor se trata de un parche adicional que

sobrescribe algunos datos del juego y añade nuevos para permitir al agente poder comunicarse con el juego.

El juego también alberga una interfaz gráfica, mediante la cual se puede navegar entre los menús del juego y ejecutar las carreras. En la figura 4.2 se puede apreciar la arquitectura interna del módulo del juego, mientras que en las figuras 4.3 y 4.4 se muestran algunas de las pantallas de la interfaz del juego.

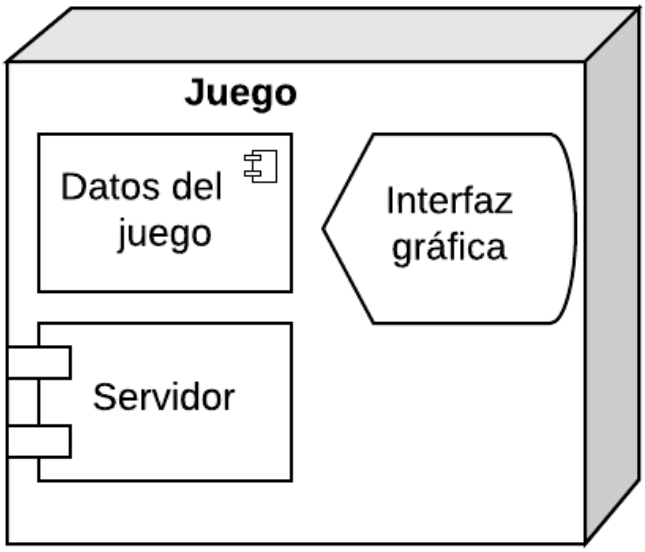

Figura 4.2: Arquitectura del juego (Fuente: elaboración propia)

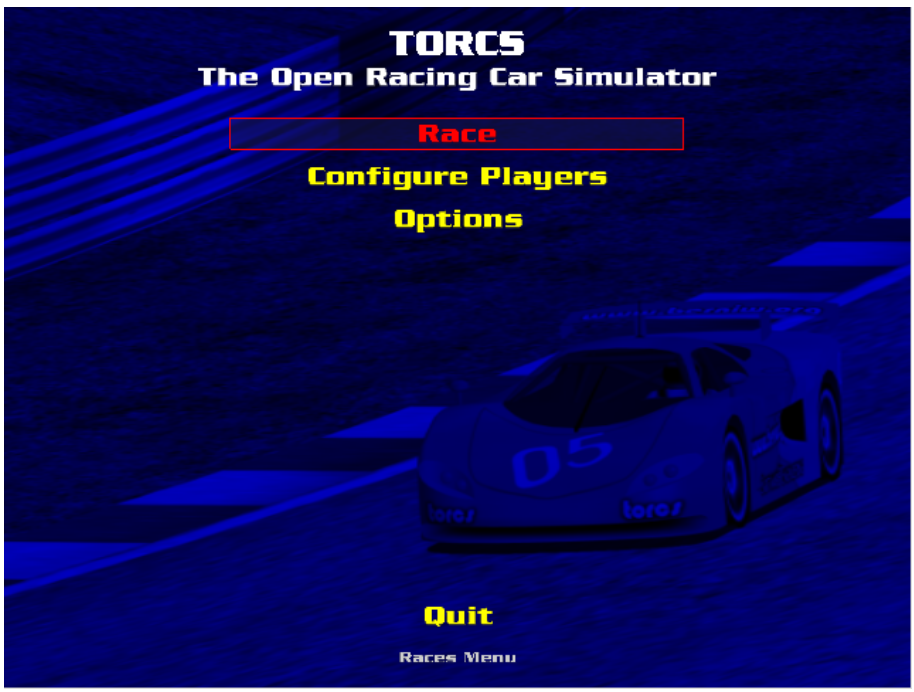

Figura 4.3: Menú principal del juego (Fuente: videojuego TORCS)

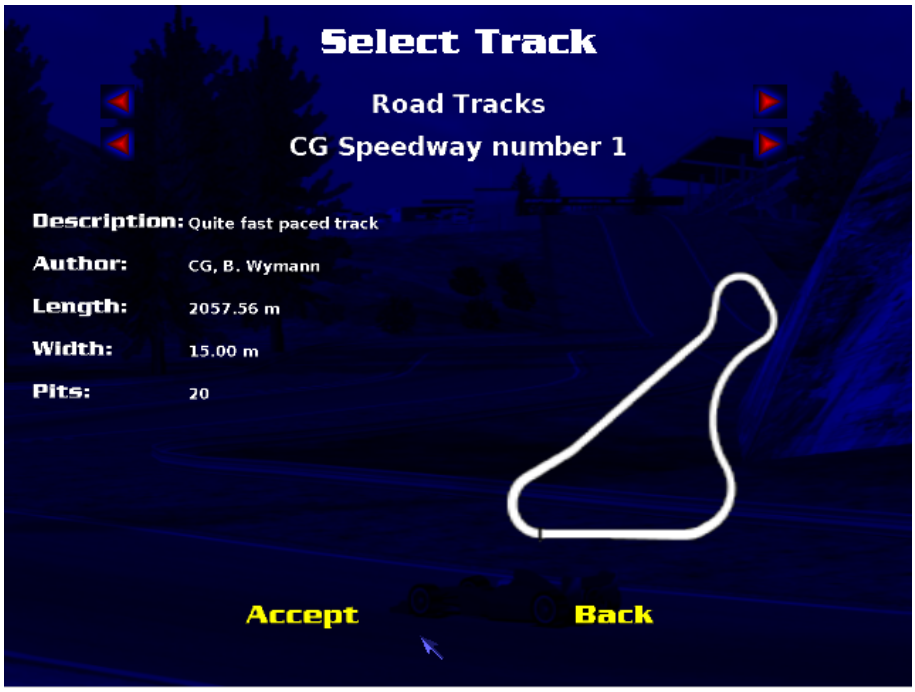

Figura 4.4: Menú de configuración de carrera (Fuente: videojuego TORCS)

Cabe destacar que todo este módulo ha sido obtenido a través de la página oficial del videojuego TORCS y que por ello ninguna parte ha sido diseñada y modificada, simplemente se comenta en este apartado para que se conozca al completo la arquitectura donde se trabaja para desarrollar el agente.

4.3.2. Agente

Se trata de un módulo adicional que se desarrolla para ejecutarlo junto con el juego y hacer que controle a uno de los vehículos del mismo. Es el módulo en el cual sí se ha aplicado un diseño definido, aunque ciertas partes se han obtenido de fuentes de terceros. Este módulo se puede dividir en dos submódulos: el cliente y la máquina de estados.

Cliente

El cliente se trata de un conjunto de clases que implementan los actores y sensores del agente. Este submódulo ha sido obtenido de la comunidad del TORCS y modificado para adaptarlo al sistema que se desarrolla. Los componentes que forman el cliente son: `Action`, `Client`, `Controller`, `MessageBasedSensorModel`, `MessageParser`, `SensorModel`, `SocketHandler`. Adicionalmente, se pueden encontrar las clases en las cuales se implementan los agentes que controlan la lógica del vehículo. En este caso se tienen dos ejemplos: `DeadSimpleSoloController` y `SimpleDriver`.

- `Action`
  Mediante esta clase se definen las acciones que se quieren enviar al servidor para que el cliente pueda ejecutarlas en el juego. Estas acciones son: acelerar, frenar, embragar, indicar la marcha, indicar la dirección, reiniciar la carrera, lectura de los ángulos. Esta clase se compone de dos métodos:

- `toString`: genera el mensaje de la acción en el formato necesario para que lo entienda el servidor.
- `limitValues`: establece los límites de los valores de cada una de las acciones.

- **`Client`**
  Se trata de la clase principal del cliente (por ello se llama igual). Es la clase encargada de establecer la conexión entre el cliente y el servidor, de cargar el controlador del vehículo y de controlar el bucle de ejecución del agente mientras dure la carrera. Se compone de tres métodos:

  - `main`: es el método principal. Se encarga de las acciones principales del cliente mencionadas anteriormente.
  - `parseParameters`: se utiliza para analizar sintácticamente los argumentos de entrada en la ejecución, comprobar sus valores y adaptarlos al formato del agente.
  - `load`: con este método se carga el controlador del vehículo dependiendo del tipo de controlador que se halla indicado con los argumentos de ejecución del cliente.

- **`Controller`**
  Esta clase sirve como base para los controladores que se quieran implementar. Incluye un enumerado para controlar el estado en las competiciones. Los métodos que componen esta clase son:

  - `intiAngles`: inicia el valor de los ángulos en los que se quiere leer las distancias.
  - `getStage`: devuelve el valor del estado de la competición.
  - `setStage`: da un valor al estado de la competición.
  - `getTrackName`: devuelve el valor del nombre de la pista donde se compite.
  - `setTrackName`: da un valor al nombre de la pista donde se compite.
  - `control`: se define como cabecera de método para implementarlo en cada controlador como se quiera. Será el método de control de los sensores del vehículo.
  - `reset`: se define como cabecera de método para implementarlo en cada controlador como se quiera. Sólo es utilizado si se realizan competiciones.

- **`SensorModel`**
  Este componente se define como una interfaz con las cabeceras de los métodos para obtener la lectura de los sensores del vehículo.

- **`MessageParser`**
  Mediante esta clase se analiza el mensaje recibido desde el servidor y se divide en *tokens* para poder acceder a cada uno de ellos de forma individual y así entender de una forma clara cada uno de los sensores del vehículo. Está compuesta por cuatro métodos:

  - `MessageParser`: es el constructor de la clase. Mediante éste se realiza todo el análisis sintáctico del mensaje y se guardan los *tokens* en una tabla *hash*.

- `printAll`: permite imprimir los *tokens* de la tabla *hash*.
- `getReading`: devuelve el valor asociado a un *token*.
- `getMessage`: devuelve el mensaje que se haya guardado en la variable de la clase.

- **`MessageBasedSensorModel`**
  En esta clase se implementan los métodos de `SensorModel` teniendo como base el análisis sintáctico que se ha hecho previamente en la clase `MessageParser`. Esta clase contiene una implementación para obtener el valor de cada uno de los sensores explicados en la tabla 8.2 además de un método que ha sido modificado para poder obtener el mensaje recibido del servidor una vez ya ha sido analizado y dividido en *tokens*.

- **`SocketHadnler`**
  Se trata de la clase que permite comunicarse con el servidor. A través de ésta se envían y reciben los mensajes. Esta clase se compone de cuatro métodos:

  - `SocketHandler`: es el constructor de la clase. Inicializa el *socket* con los valores de la dirección IP y el puerto.
  - `send`: envía un paquete de datos al servidor.
  - `receive`: recibe un paquete de datos del servidor. Existe una versión del mismo método que tiene en cuenta un *tiemout* para tener en cuenta el tiempo que tarda en recibir alguna respuesta del servidor.
  - `close`: cierra el *socket*.

- **`DeadSimpleSoloController`**
  Esta clase es un agente que viene ya implementado. Se trata de un controlador muy simple que intenta ajustar la velocidad a un valor máximo preestablecido (15 por defecto) y que no cambia de marchas. Además, el control de la dirección es demasiado básico, por lo que en los giros tiene ciertos problemas.

- **`SimpleDriver`**
  Se trata de otro de los controladores que vienen implementados con el cliente descargado. La complejidad del agente es mucho mayor que la de *DeadSimple-SoloDriver*. El controlador dispone de variables finales definidas para tener una mayor exactitud en la lógica del agente. Además del método principal de control, se implementan nuevos métodos para obtener una mayor precisión en los cálculos de la acción. Estos métodos son:

  - `getGear`: a partir del valor de la marcha y las RPM que devuelven los sensores y los valores preestablecidos en las variables finales, permite un cálculo ajustado del momento necesario para cambiar de marcha.
  - `getSteer`: calcula la dirección del coche en función de la posición del coche en la pista y de los ángulos hacia los que está enfocado el vehículo.
  - `getAccel`: calcula si deber acelerar o no en función de la velocidad actual, la proximidad de objetos en los ángulos más cerrados y si se encuentra dentro o fuera del trazado.
  - `filterABS`: simula el funcionamiento del ABS en las frenadas fuertes.

- `clutching`: ajusta el valor del embrague según la situación (inicio de carrera o no) y la marcha en la que se encuentra.
- `initAngles`: inicializa los ángulos de visión del vehículo de forma diferente a cómo lo hace el controlador por defecto.

Mediante el método principal de control, a parte de estos métodos que se acaban de mencionar, también maneja la situación si el vehículo está estancado en algún punto del circuito.

A continuación, en la figura 4.5 se representa la interacción que hay entre las diferentes clases y los métodos de estas para conseguir que el agente funcione de forma adecuada.

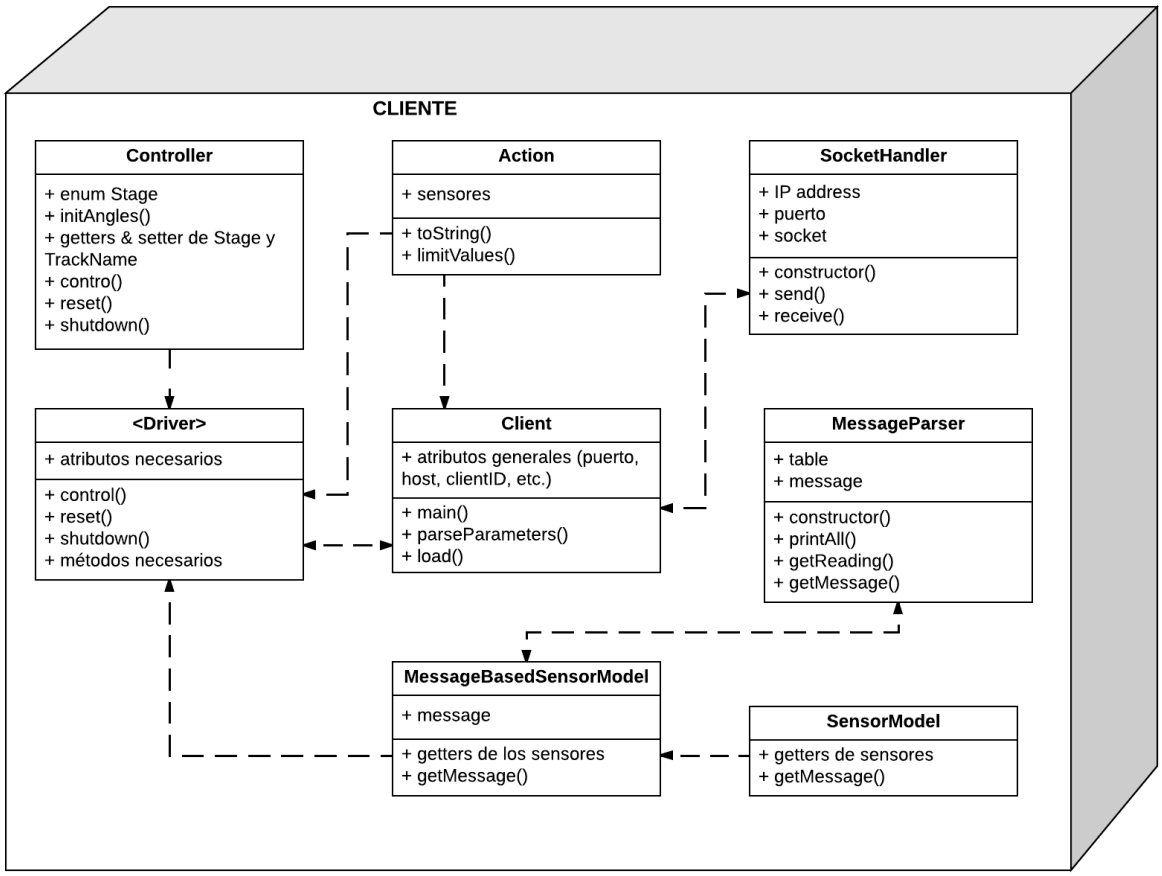

Figura 4.5: Diagrama de componentes (Fuente: elaboración propia).

Cabe destacar, que el tipo de conexión entre el cliente y el servidor sigue el protocolo UDP. Mediante este protocolo, no es necesario que el cliente intente establecer un conexión con el servidor como se hace con TCP. Basta con indicar la dirección IP en la que se encuentra dicho servidor y el puerto por el que va a escuchar. La principal ventaja que aporta es la velocidad de transmisión al no tener un control de confirmación de recibo de paquetes, lo que evita retrasos e interrupciones en la ejecución de la simulación. Pero esta misma característica significa una desventaja ya que los paquetes pueden llegar en un orden que no es el mismo en el que se enviaron. En este caso no resulta ningún problema ya que la velocidad con la que se envían y reciben los paquetes es tan alta que no se aprecian claros errores en las acciones que realiza el vehículo.

MEF

En este apartado se procede a explicar los diferentes diseños de máquinas de estados finitos que se han realizado durante el desarrollo del proyecto para conseguir implementar un agentes automático que cumpla el objetivo del trabajo.

Se han diseñado dos MEF's diferentes como posibles soluciones para el trabajo: en una se aplica el concepto de navaja de Ockham y se intentan simplificar el sistema dividiéndolo en máquinas de estados finitos muy sencillas, y en la otra se realiza un diseño completo unificando todos los procesos y estados en una única MEF.

Navaja de Ockham

Como se mencionaba anteriormente, la primera idea que se tuvo fue simplificar el sistema, diseñar un modelo de máquinas de estados sencillas que se encargarse cada una de ellas de una de las facciones del vehículo. De esta forma se obtienen 3 MEF's simples que se muestran en las imágenes 4.6, 4.7 y 4.8.

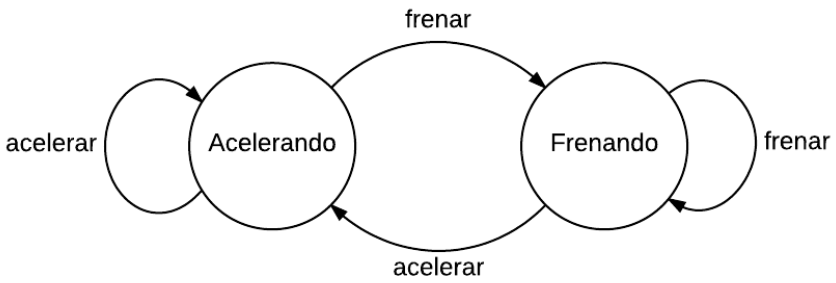

Figura 4.6: MEF que controla la velocidad (Fuente: elaboración propia).

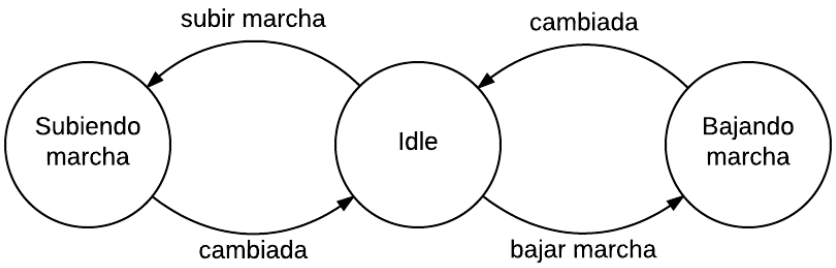

Figura 4.7: MEF que controla las marchas (Fuente: elaboración propia).

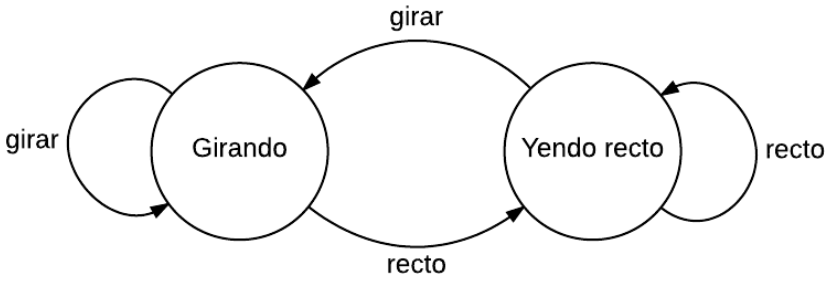

Figura 4.8: MEF que controla la dirección (Fuente: elaboración propia).

- **MEF de velocidad.** Se definen 2 estados: *acelerando* y *frenando*. En cada uno de ellos se especifica el nivel de aceleración o de frenada necesarios evaluando la situación. No es necesario un tercer estado en el que ni se acelera ni se frena debido a que el nivel de aceleración puede tomar valor 0 y es equivalente a esta situación descrita. Las transiciones son *acelerar* y *frenar*. Estas transiciones dependen de la lógica del agente. En este caso se tienen en cuenta los ángulos de visión del coche y la velocidad actual del mismo.
- **MEF de marchas.** Se definen 3 estados: *idle*, *subiendo marcha* y *bajando marcha*. En este caso sí es necesario el estado *idle* debido a que no puedes quedarte subiendo o bajando marcha todo el rato. Las transiciones entre estados se describen con *subir marcha*, *bajar marcha* y *cambiada*. Para la toma de decisiones de estas transiciones se utilizan las RPM actuales del vehículo y unos valores predefinidos de cambio de marcha para cada una de ellas. En la interacción siguiente, el estado del juego nos permitirá saber si la marcha ha cambiado. En el caso de que esto no pase, se le seguirá diciendo al juego la marcha que se desea cambiar hasta que ocurra.
- **MEF de dirección.** Se definen 2 estados: *yendo recto* y *girando*. En el estado *girando* no hace falta conocer la dirección en la que se gira, puesto que el valor del sensor del coche tiene un rango de -1 a 1, donde los valores negativos significan girar hacia la izquierda y los positivos hacia la derecha[3]. Las transiciones entre ambos estados son *girar* y *recto* las cuales se obtienen tras computar el valor de la dirección del coche. Para evaluar la dirección se tienen en cuenta los ángulos de visión del vehículo, así como la velocidad actual y algunas variables de control predefinidas.

Modelo unificado

La segunda idea que se ha diseñado presenta una única máquina de estados en la cual se controlan todas las condiciones a la vez. Este modelo resulta más lioso y complejo que el anterior ya que implica un mayor número de transiciones y un control de condiciones en cada uno de los estados. La MEF que se ha diseñado se puede observar en la figura 4.9.

[3]Existe una variación de esta MEF en la que sí se tienen los estados especificando la dirección de giro en la cual las transiciones evalúan si las distancias de los ángulos son mayores en un lado o en otro. Esta variación se implementa en la segunda solución propuesta.

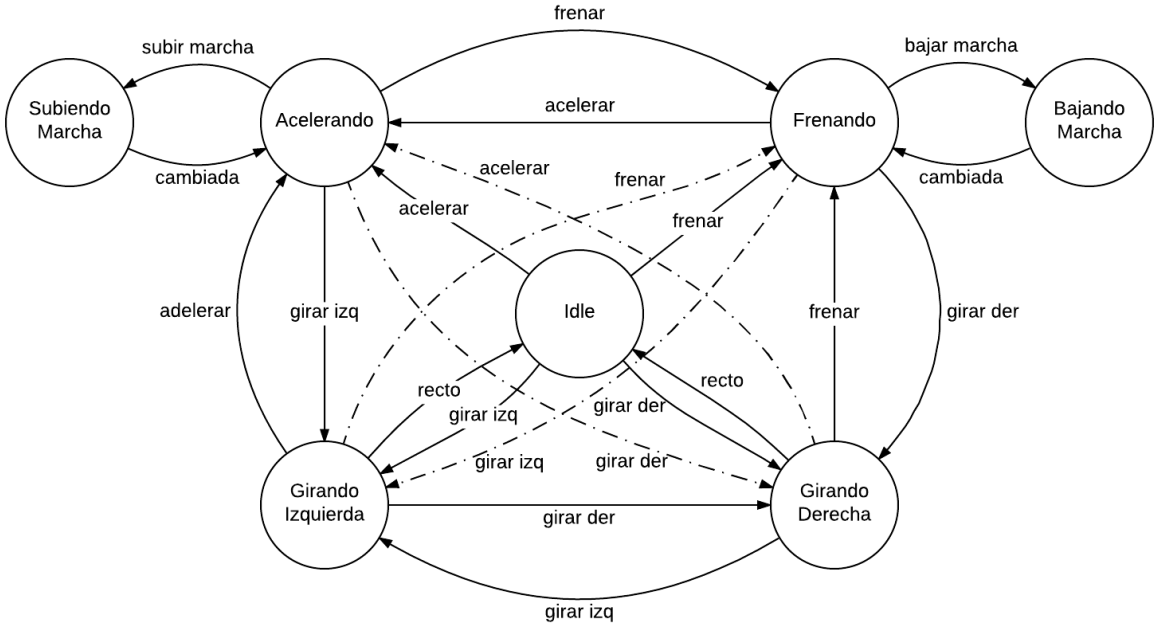

Figura 4.9: MEF que unifica todos los estados (Fuente: elaboración propia).

Como se puede observar, la máquina de estados finitos queda muy compleja por la gran cantidad de transiciones que existen. En los estados que controlan la velocidad del vehículo es necesario saber si hay que cambiar de marcha o de dirección, mientras que en los estados que verifican la dirección es necesario tener un control de la velocidad.

Algunas de las transiciones se pueden omitir estableciendo condiciones con prioridad. Por ejemplo, en el estado *girando izquierda*, podríamos necesitar cambiar de marcha, pero la transición no existe. Este problema se puede solucionar estableciendo la acción de controlar la velocidad con una mayor prioridad (así transitará primero a estos estados), después se ajustaría la prioridad del cambio de marcha y por último el control de la dirección. De esta forma, estando girando a la izquierda y acelerando (se transita entre ambos estados de forma alterna), si nos encontramos en el estado *girando izquierda* y el sistema detecta que es necesario cambiar de marcha, primero transitará al estado *acelerar* (ya que se estaba acelerando) y a continuación cambiará de marcha. Una vez que haya subido de marcha, las transiciones volverán a transcender de la misma forma que ocurría antes (alternando entre acelerar y girar).

Este modelo de máquina de estados finitos presenta alguna deficiencias frente al modelo descrito anteriormente, pero estas características se comentan en la siguiente sección.

4.4. Solución escogida

En la sección anterior (4.3) se ha presentado la arquitectura del sistema con dos diseños que presentan una posible solución al problema planteado. Estas opciones son:

- Varias máquinas de estados finitos simples que se coordinan.
- Una única máquina de estados finitos compleja.

Comparando una solución con la otra, se puede extraer como principal característica la ventaja que presenta la simplicidad (principio de la navaja de Ockham), ya que permite entender el modelo mucho mejor. Pero esa no es la única ventaja. Con el modelo simple, se pueden controlar varias funcionalidades a la vez ya que dispones de una MEF para cada una de ellas, mientras que con el modelo complejo pierdes efectividad en este tema. Por otro lado, a la hora de tener que implementar los diseños en código, resulta más legible y cómodo utilizar el primero de ellos. Es por estas características por las que se ha decidido tomar el primer modelo como solución al problema.

Una vez que se ha decidido cuál es la solución final que se va a utilizar en el proyecto, se puede especificar la arquitectura final del sistema. Teniendo en cuenta todo lo que se ha comentado anteriormente, se puede entender que la arquitectura del sistema sigue el modelo de una *blackboard* (pizarra). Este tipo de modelo está compuesto por múltiples elementos o agentes (clases del cliente + MEF) que controlan diferentes tareas y que se comunican y cooperan entre ellos para alcanzar los objetivos planteados. Mediante la pizarra, se facilita la comunicación interna entre elementos y externa con el "mundo". En la figura 4.10 se puede observar la arquitectura final con el uso de la *blackboard* como el concepto base.

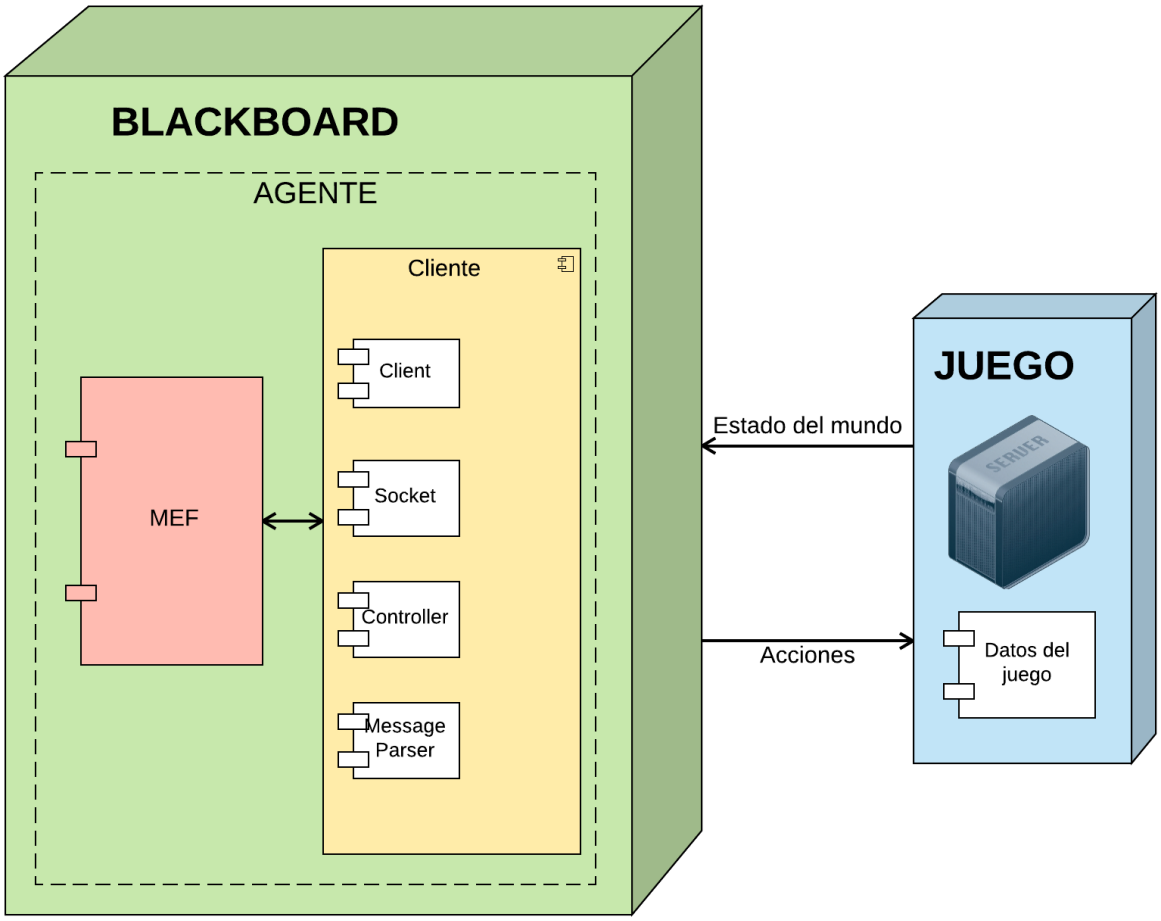

Figura 4.10: Arquitectura final del sistema (Fuente: elaboración propia).

La pizarra controla la coordinación entra las diferentes máquinas de estados finitos así como los sensores del vehículo y las acciones que se envía al servidor. Además, ofrece una comunicación entre las MEF's y los sensores para poder establecer las condiciones de transición entre estados.

Capítulo 5

Implementación del sistema

En este capítulo del documento se procede a explicar el proceso de implementación del sistema, así como las consiguientes descripciones del código que se ha usado. Para llevar a cabo este proceso, se toman como base los casos de uso y requisitos del capítulo 3 y el modelo del diseño y solución final descritos en el capítulo 4.

Es necesario tener en cuenta que ha habido partes del código que se han extraído de terceras personas y que han sido estudiadas y adaptadas al proyecto. Las clases y métodos principales de este código han sido brevemente explicados en el capítulo 4, lo que no impide que se realice una explicación en caso necesario en este punto del documento.

A continuación, se procede a describir la implementación de las máquinas de estados finitos.

5.1. MEF's

Antes de proceder a explicar el código que se ha implementado, se va a comentar cómo se han incorporado las máquinas de estados al cliente que ya se disponía.

Dado que Java no contiene ningún paquete que implemente alguna clase para crear máquinas de estados, es necesario crear las clases necesarias. Otra forma de acoplar un sistema de máquinas de estados al código, sería importando clases que ya estén definidas por terceras personas y reutilizar ese código. En este caso, se ha optado por tomar como referencias diferentes ejemplos de distintas implementaciones de MEF's en Java y crear desde cero las clases necesarias para el desarrollo del proyecto. De esta forma, se han obtenido 2 clases principales: `StateMachine` y `State`; y una interfaz: `ActionHandler`.

5.1.1. StateMachine

Se trata de la clase mediante la cual se definen las máquinas de estados que se vayan a implementar. Está compuesta de los siguientes atributos:

- `id`: es un identificador mediante el cual se le da un nombre a la MEF. Su funcionalidad es meramente informativa.

- `states`: se trata de una lista en la que se contienen todos los estados que forman la MEF. Sus componentes son instancias de la clase `State`.
- `curr_S`: indica el estado actual en el que se encuentra la máquina. También es una instancia de la clase `State`.

Los métodos que dan funcionalidad a esta clase son:

- `StateMachine`: se trata del constructor de la clase. Inicializa una máquina de estados con el identificador que se le pasa por parámetros. La lista de estados también se inicializa de forma vacía.
- `addState`: permite añadir una nueva instancia de estado a la MEF. Se comprueba si el estado ya existe en la lista, en el caso negativo se añade. Si no se ha podido añadir el estado, se indica al sistema el error sucedido. Si no existe ningún estado actual, el nuevo estado se indica como tal.
- `move`: método que indica al estado actual a qué estado tiene que transitar. Se comprueba si el estado al que se transita existe en la MEF; si no, se indica el error y se aborta la transición. Si se puede transitar, el estado actual (`curr_S`) cambiará al nuevo estado.
- `getCurrent`: devuelve el estado actual.
- `setCurrent`: permite asignar cuál es el estado actual.
- `getStates`: devuelve la lista de estados de la MEF.
- `cleanMachine`: busca los estados inalcanzables y los elimina.

Mediante esta clase, se pueden implementar las 3 máquinas de estados finitos que se han descrito en el capítulo 4. De esta forma, se obtiene el siguiente código:

```
StateMachine sm_speed = new StateMachine("MEF_Speed");
StateMachine sm_gear = new StateMachine("MEF_Gear");
StateMachine sm_steer = new StateMachine("MEF_Steer");
```

Los estados y transiciones que definen la misma, se añaden después. En el siguiente apartado se explica cómo.

5.1.2. State

Con esta clase se pueden implementar los estados y transiciones que componen a una MEF. Esta clase contiene los siguientes atributos:

- `name`: nombre identificativo del estado. Su funcionalidad es meramente informativa.
- `transitions`: se trata de una tabla *Hash* mediante la cual se definen las transiciones de este estado con otros. Se utiliza este tipo de variable para poder asociar una acción a cada transición. De esta forma, se tiene un dupla estado, acción que forma la transición.

La funcionalidad de esta clase está definida por los siguientes métodos:

- `State`: es el constructor de la clase. Inicializa un estado con el nombre que se indica por parámetros. También se inicializa la tabla de transiciones vacía.
- `addTransition`: permite añadir una nueva transición al estado. Se comprueba que la nueva transición no exista buscando si el estado que se recibe por parámetros ya pertenece a la tabla. En el caso de que no pertenezca, se añade; en caso erróneo se aborta la creación de la transición y se informa del error.
- `move`: permite realizar la transición y lanzar la acción asociada. Este método es llamado desde el método `move` de la máquina de estados. Se comprueba que el estado que se recibe por parámetros pertenezca a la tabla de transiciones. En caso correcto, se activa la acción indicada en dicha transición y se indica a la máquina de estados que la transición se ha realizado con éxito. En caso de fallo, se indica a la máquina de estados que no se ha podido transitar.

A través de esta clase, se pueden generar los estados y las transiciones y, de esta manera, añadirlos a las máquinas de estados correspondientes. Un ejemplo de código es el siguiente:

```
State accel = new State("Accel");
State brake = new State("Brake");
accel.addTransition(brake, <ActionHandler>);
sm_speed.addState(accel);
sm_speed.addState(brake);
```

5.1.3. ActionHandler

La mayoría de las transiciones (por no decir todas) van acompañadas de la ejecución de una acción. Por ello, es necesario tener un manejador que ejecute dicha acción cuando se realiza una transición en la máquina de estados. Esta interfaz permite crear los manejadores de cada acción que se realiza en el agente. Por ello, sólo se compone de un método:

- `action:` controla la acción asociada a la transición. Se puede implementar de dos formas diferentes: creando en el controlador los métodos con las acciones y la lógica necesarios y llamándolos desde este método, o incluyendo directamente la lógica en el método y liberando al controlador de ello.

En los casos en los que se ha tomado el cliente que ya estaba implementado, se ha optado por la primera opción. Pero si se desea implementar un nuevo controlador, es recomendable utilizar la segunda y, en el manejador, modificar los valores de los sensores del agente.

Un ejemplo sencillo de un manejador sería el siguiente:

```
public class ActionAccel implementes ActionHandler {
   public void action(Client c) {
      c.accel();
   }
}
```

En este ejemplo que se muestra, se da por echo que existe una función `accel` en el cliente que calcula el nivel de aceleración del vehículo. De esta forma, en el agente se tendría una instancia de esta clase de la forma que sigue:

```
ActionAccel act_Accel = new ActionAccel();
accel.addTransition(brake, act_Accel);
```

5.2. Agente

En esta sección se procede a explicar los detalles que faltan para comprender cómo se ha implementado el agente automático del proyecto.

Una vez que ya se tienen todas las clases necesarias (las que venían incluidas con el cliente y las que se han tenido que crear), se puede iniciar la implementación del agente que se desea desarrollar. En este proyecto, se ha aprovechado la existencia de dos agentes o controladores para poder realizar pruebas de comportamiento con ellos. De esta forma, en el controlador `SimpleDriver` se han implementado las máquinas de estados que se han escogido como solución final del proyecto (capítulo 4) reutilizando así las funciones y variables de control de los sensores que este controlador tiene. En el diagrama de flujo de la figura 5.1 se puede observar el proceso que sigue el agente durante la ejecución de juego.

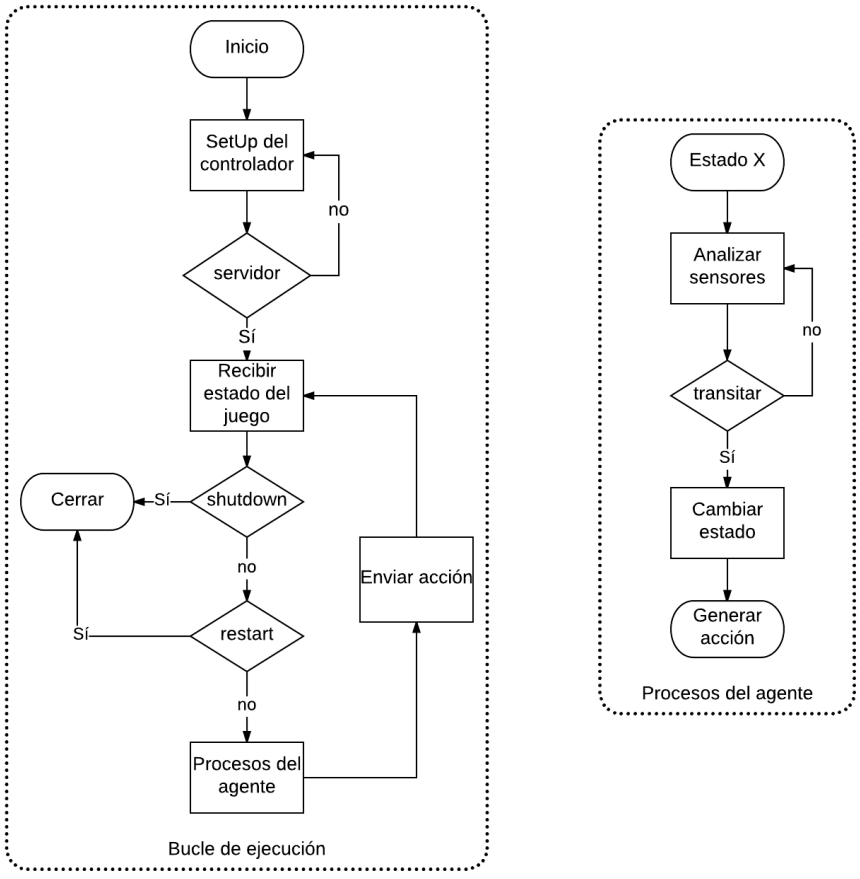

Figura 5.1: Diagrama de flujo del agente `SimpleDriver` (Fuente: elaboración propia).

En añadido, la figura 5.2 muestra la interacción que se produce entre las acciones que surgen de las transiciones de las máquinas de estados finitos y las funciones de control de sensores del controlador.

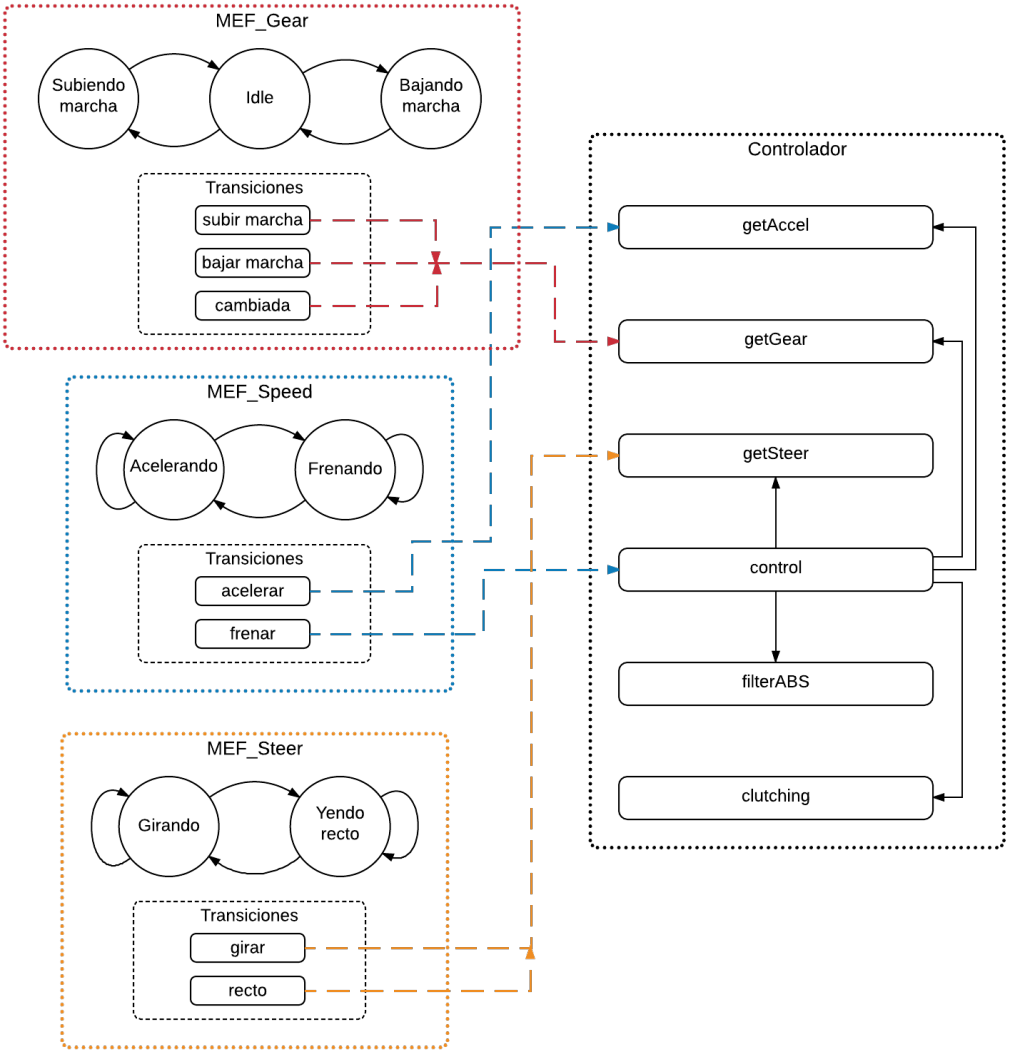

Figura 5.2: Interacción entre las MEF's y el controlador `SimpleDriver` (Fuente: elaboración propia).

Como se ha explicado anteriormente, se puede observar que al producirse una transición, se genera una acción la cuál está representada en el controlador mediante alguna de las funciones de control de los sensores y de cálculo de valores.

Otra de las opciones sería eliminar la funciones de control de sensores del controlador e incluir dicha funcionalidad en la acción del manejador. Siguiendo este modelo, sería necesario implementar un manejador para cada una de las acciones. En el caso del `SimpleDriver` se tendría un manejador para cada una de las siguientes funciones de control:

- **Nivel de aceleración.**

  ```
  public class ActionAccel implementes ActionHandler
  ```

- **Nivel de freno.**

  ```
  public class ActionBrake implementes ActionHandler
  ```

- **Valor de la dirección.**

  ```
  public class ActionSteer implementes ActionHandler
  ```

- **Caja de cambios.**

  ```
  public class ActionGear implementes ActionHandler
  ```

- **ABS en las frenadas.**

  ```
  public class ActionABS implementes ActionHandler
  ```

- **Nivel de embrague.**

  ```
  public class ActionClutching implementes ActionHandler
  ```

Alguna de las funcionalidades anteriores pueden ser combinadas ya que podría ser necesario ejecutarlas tras la misma transición. Esta combinación se puede realizar modificando levemente el código, obteniendo de esta manera menos manejadores.

Capítulo 6

Resultados y evaluación

En este capítulo de la memoria se muestran las pruebas que se han llevado a cabo para comprobar el correcto funcionamiento del sistema teniendo en cuenta la especificación de requisitos realizada en el capítulo 3. Además, se comentarán los resultados que se han obtenido tras la implementación del agente.

Las pruebas siguen el formato especificado en la tabla 6.1.

Tabla 6.1: Plantilla pruebas

Identificador: P-XX	
Descripción	
Requisitos verificados	
Resultado esperado	
Evaluación	

A continuación se describe el significado de cada una de las entradas de la tabla:

- **Identificador:** representación unívoca de la prueba. Los caracteres que lo identifican son:
  - P. Prueba.
  - XX. Número de prueba.
- **Descripción:** explicación de la prueba.
- **Requisitos verificados:** requisitos que se cubren con la prueba.
- **Resultado esperado:** producto que se espera al realizar la prueba.
- **Evaluación:** indica si la prueba ha sido superada con éxito.

De la tabla 6.2 a la tabla 6.7 se describen las pruebas que se han realizado para la verificación del sistema.

Tabla 6.2: P-01

Identificador: P-01	
Descripción	Aceleración del vehículo
Requisitos verificados	RF-01
Resultado esperado	El vehículo acelera durante la carrera
Evaluación	Positiva

Tabla 6.3: P-02

Identificador: P-02	
Descripción	Frenado del vehículo
Requisitos verificados	RF-02
Resultado esperado	El vehículo frena durante la carrera
Evaluación	Positiva

Tabla 6.4: P-03

Identificador: P-03	
Descripción	Cambio de marchas
Requisitos verificados	RF-03 RF-04 y RF-08
Resultado esperado	El vehículo cambia de marcha durante la carrera mientras acelera o frena
Evaluación	Positiva

Tabla 6.5: P-04

Identificador: P-04	
Descripción	Control de la dirección del vehículo
Requisitos verificados	RF-05 y RF-06
Resultado esperado	El vehículo cambia de dirección durante la carrera
Evaluación	Positiva

Tabla 6.6: P-05

Identificador: P-05	
Descripción	Comunicación entre las máquinas de estados y el controlador
Requisitos verificados	RF-09
Resultado esperado	Las máquinas de estados utilizan los sensores del controlador y el controlador activa sus acciones según los estados
Evaluación	Positiva

Tabla 6.7: P-06

Identificador: P-06	
Descripción	Coordinación entre las diferentes máquinas de estados controlando el agente
Requisitos verificados	RF-10
Resultado esperado	Las máquinas de estados finitos son capaces de controlar los diferentes factores y acciones del agente al mismo tiempo durante una carrera
Evaluación	Positiva

El requisito funcional RF-07 (tabla 3.18) puede ser verificado con cualquiera de las pruebas de 6.2 a 6.7, ya que para conseguir pasar cualquiera de estas pruebas es necesario recibir y enviar información al servidor.

Por otro lado, los requisitos no funcionales de este sistema no son aplicables a ninguna prueba de software posible, pero pueden verificarse observando la solución aplicada.

En la tabla 6.8 se muestra la matriz de trazabilidad entre los requisitos funcionales del sistema y las pruebas realizadas. De esta forma, se puede comprobar de una manera dinámica qué prueba cubre cada requisito.

Tabla 6.8: Matriz de trazabilidad

	P-01	**P-02**	**P-03**	**P-04**	**P-05**	**P-06**
RF-01	■					
RF-02		■				
RF-03			■			
RF-04			■			
RF-05				■		
RF-06				■		
RF-07	■	■	■	■		
RF-08			■			
RF-09					■	
RF-10						■

Para concluir este capítulo, se puede mencionar que los resultados obtenidos son positivos y satisfactorios ya que el sistema es capaz de superar todas las pruebas definidas, cubriendo de esta manera los requisitos que se especificaron en el capítulo 3 de este documento. Esta comprobación se puede observar en la matriz de trazabilidad (tabla 6.8).

Capítulo 7

Planificación del trabajo, presupuesto y entorno socio-económico

En este capítulo del documento se recogen la planificación del proyecto, los costes del mismo y el entorno socio-económico al que afecta.

7.1. Planificación

Para definir una buena planificación, es necesario especificar la metodología software que se utiliza durante el proyecto. En este caso se ha optado por un desarrollo o modelo en cascada. Se ha escogido este tipo de metodología ya que es el que mejor se ajusta a la forma en la que se quiere tratar el trabajo. Al tratarse de un proyecto en el que se va a investigar y probar diferentes configuraciones en el software, con esta metodología se asegura un desarrollo correcto y que permite modificar lo que se haya implementado, así como corregir los posibles errores. El modelo en cascada se divide en varias fases que se puede observar en la figura 7.1.

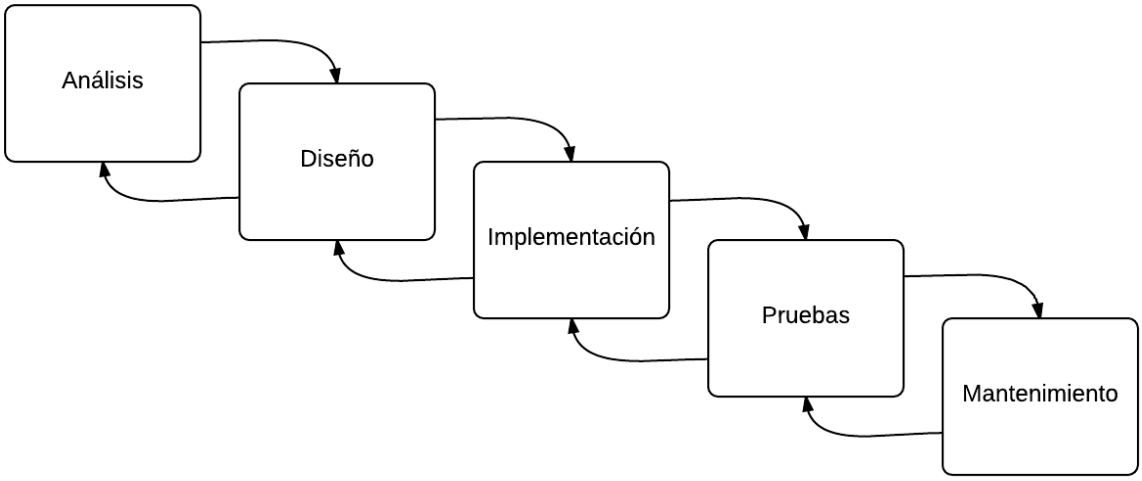

Figura 7.1: Modelo en cascada (Fuente: elaboración propia).

- **Análisis.** En esta fase es necesario asentar las bases del producto final. Se realizan reuniones con el tutor para especificar los requisitos y casos de uso del proyecto. También se incluye un proceso de investigación del estado del arte para poder abordar el trabajo de la mejor forma posible.
- **Diseño.** Una vez establecidos los requisitos y casos de uso, se procede a realizar un diseño de los componentes del proyecto. En esta fase se busca obtener un descripción de estos componentes o módulos que permitan, después, llevar a cabo la implementación de los mismo.
- **Implementación.** Con el diseño terminado, se procede a implementar en código fuente los módulos que se han definido.
- **Pruebas.** Es necesario comprobar que la implementación realizada cubre los requisitos y casos de uso que se definieron en el análisis del sistema. En esta fase se describen y llevan a cabo las pruebas necesarias para asegurar dicha cobertura.
- **Mantenimiento.** A pesar de que se hayan pasado todas las pruebas, en ocasiones pueden surgir errores que necesiten ser solucionados, por ello se realiza una fase de mantenimiento del sistema.

Como se puede observar en la figura 7.1, la metodología que se ha seguido permite volver de una fase a la anterior o anteriores si fuese necesario modificar algo. Este sistema permite realizar mejoras de forma sencilla y eficaz, además de corregir los errores de una manera guiada.

Una vez escogida y definida la metodología que se va a seguir durante el desarrollo del proyecto, se especifica la planificación del mismo. Por ello, a continuación, con la ayuda de tablas y diagramas, se presenta la planificación que se ha seguido en el progreso de este trabajo.

En primer lugar, se muestra una duración[1] general del proyecto (tabla 7.1).

Tabla 7.1: Duración general del proyecto

Trabajo	**Duración**	**Fecha de comienzo**	**Fecha de finalización**
Desarrollo del agente automático	14	02/05/2017	06/09/2017
Documentación	2	07/09/2017	21/09/2017

En la tabla 7.1 se puede observar en la fila que hace referencia al "Desarrollo del agente automático" que la duración entre la fecha de inicio y la fecha de finalización (18 semanas) es mayor que la duración indicada (14 semanas). Esto es debido al periodo vacacional de 4 semanas que hubo durante el mes de agosto en el cual no se produjo ningún tipo de trabajo para el proyecto.

Una vez definida la duración general del proyecto, se desglosa la duración de cada etapa vista anteriormente en la descripción de la metodología de software. Este desglose se encuentra documentado en la tabla 7.2.

[1]Las duraciones de los procesos del proyecto se especifican en semanas.

Tabla 7.2: Planificación del proyecto

Tarea	Duración	Fecha de inicio	Fecha de finalización
Análisis	**3**	**02/05/2017**	**23/05/2017**
Reuniones con el tutor	2	02/05/2017	16/05/2017
Periodo de investigación	2	09/05/2017	23/05/217
Diseño	**5**	**23/05/2017**	**27/06/2017**
Diseño de MEF	3	23/05/2017	13/06/2017
Diseño del agente	3	06/06/2017	27/06/2017
Implementación	**6**	**27/06/2017**	**08/08/2017**
Adaptación del cliente	2	27/06/2017	11/07/2017
Implementación de MEF	4	12/07/2017	08/08/2017
Pruebas	**2**	**06/09/2017**	**20/09/2017**
Especificación de pruebas	1	06/09/2017	12/09/2017
Ejecución de pruebas	1	13/09/2017	20/09/2017

A partir del desglose de la tabla 7.2 se ha realizado un diagrama de Gantt en el que se muestra de forma gráfica toda la planificación del proyecto. Este diagrama de Gantt queda representado en la figura 7.2.

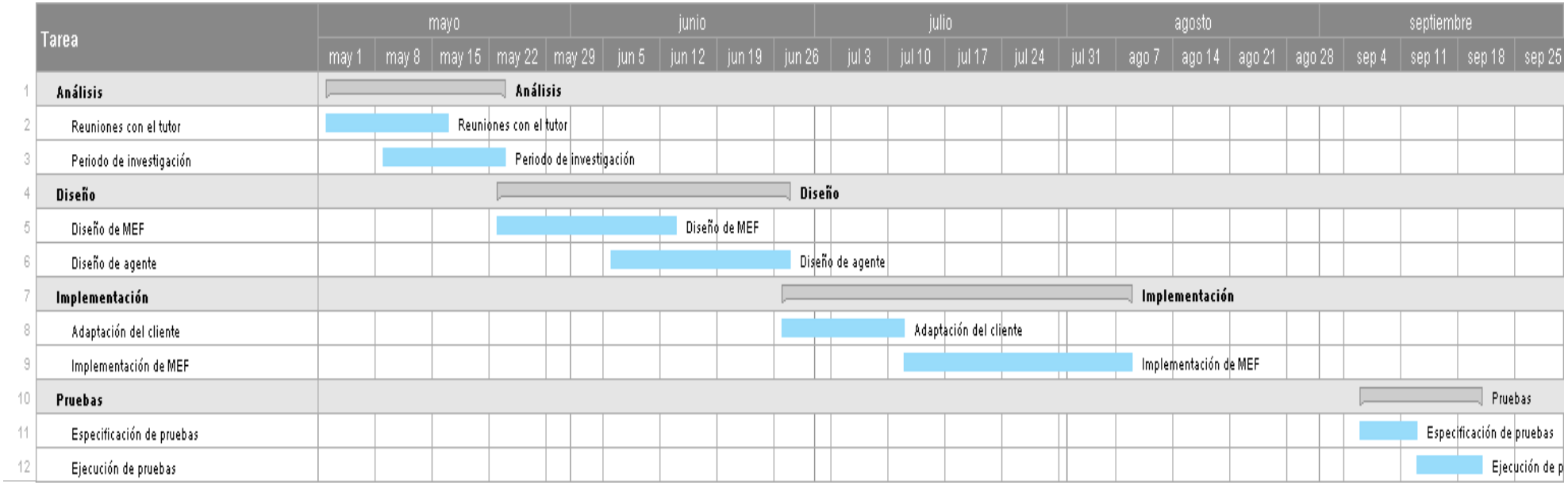

Figura 7.2: Diagrama de Gantt (Fuente: elaboración propia).

7.2. Presupuesto

En todo proyecto es importante y necesario definir los costes[2] que implicará el mismo. En esta sección se explicarán los diferentes costes que cubren este trabajo, teniendo en cuenta los costes de personal, los costes de material así como los costes indirectos.

7.2.1. Costes de personal

Para poder cuantificar los costes de personal es necesario definir los roles de las personas que trabajan en el proyecto:

- **Jefe de proyecto:** responsable de la dirección y coordinación del proyecto.
- **Analista:** encargado de la definición de los requisitos y casos de uso, así como de la investigación.
- **Diseñador:** responsable de realizar el diseño del sistema.
- **Programador:** encargado del desarrollo del código fuente del proyecto.
- **Técnico de pruebas:** responsable de realizar las pruebas del sistema.

En la tabla 7.3 se especifican los costes por hora de cada personal, así como las horas dedicadas al proyecto y los costes totales de personal.

Tabla 7.3: Costes de personal

Rol	Coste por hora	Horas de trabajo	Total
Jefe de proyecto	30	96	2880
Analista	25	108	2700
Diseñador	22	180	3960
Programador	18	288	5184
Técnico de pruebas	20	72	1440
Total	-	744	16164

7.2.2. Costes de material

A continuación, se realiza un desglose de los costes del material que se ha empleado durante la realización de este proyecto. Los materiales utilizados están definidos en la sección 1.4, pero en este apartado sólo de hará mención de aquellos que suponen algún coste al proyecto. Para especificar estos costes, se ha seguido la siguiente fórmula:

$$coste\ total = \frac{precio}{periodo\ de\ amortización} \cdot periodo\ de\ uso \tag{7.1}$$

donde:

- **coste total:** coste final del producto en el proyecto.
- **precio:** precio inicial del producto.
- **periodo de amortización**[3]**:** tiempo necesario para amortizar el precio del pro-

[2]Todos los costes del proyecto están especificados en euros.

[3]El periodo de amortización y el uso están especificados en semanas.

ducto.

- **periodo de uso:** tiempo que se empleará el producto durante el proyecto.

En la tabla 7.4 se desglosan los costes de material del proyecto.

Tabla 7.4: Costes materiales

Producto	**Precio**	**Amortización**	**Uso**	**Coste en proyecto**
PC	1000	260	16	61'54
Licencia Windows 8.1 Pro	39'99	104	16	6'15
Licencia Académica de Matlab	2000	156	1	12'82
Simulink	3000	156	1	19'23
Total	-	-	-	69'74

El resto de licencias y materiales que se han utilizado en el proyecto y no aparecen en la tabla 7.4 tienen un coste de 0€ y por ello no se añaden.

7.2.3. Costes totales

Los costes que se han especificado anteriormente hacen referencia a los costes directos del proyecto. A parte de éstos, es necesario definir los costes indirectos del mismo, así como un margen de riesgo. Además, es necesario añadir el 21 % de IVA sobre el total de todo lo anterior.

Para este proyecto se definirá un 10 % como costes indirectos y otro 15 % para el margen de riesgo. En la tabla 7.5 se especifican los costes totales del proyecto.

Tabla 7.5: Costes totales

Concepto	**Cantidad**
Costes directos	16233'74
Costes indirectos	1623'37
Margen de riesgo	1623'37
Base imponible	16480'49
IVA	3460'90
Total	**19941'39**

Este proyecto no presenta ningún beneficio. La razón de ello se explica en la sección 7.3.

7.3. Entorno socio-económico

En esta sección se analiza el entorno socio-económico que rodea al proyecto, así como el impacto que éste puede generar en dicho entorno.

Como se ha mencionado en la sección 7.2 este proyecto no genera ningún beneficio. Esto es debido a que se trata de un trabajo de investigación y no del desarrollo de una

aplicación o producto destinado a la venta. Los resultados que se obtengan del trabajo servirán como aporte a la comunidad. Es decir, el producto de este proyecto, más allá del código que se ha generado para realizar el agente automático, se trata de un conocimiento aditivo para el colectivo que engloba a los videojuegos y a la inteligencia artificial y, en mayor medida, a la sociedad.

Sí que es cierto, que en un futuro, podría tener un impacto sobre el desarrollo de inteligencia artificial, por ejemplo en progreso de la investigación de los coches autónomos o en los NPC's de algunos videojuegos. Pero todo el impacto que pueda generar se albergaría en los trabajos futuros del proyecto.

Capítulo 8

Conclusiones y trabajos futuros

Con este capítulo se alcanza, casi, el final del documento. En él se va a comentar las conclusiones extraídas tras la realización de todo el proyecto, así como algunos de los posibles trabajos futuros que se podrían llevar a cabo a partir de este.

8.1. Conclusiones

Tras el culmen de este proyecto, se extraen conclusiones técnicas sobre el mismo así como algunas conclusiones personales sobre todo el proceso.

En primer lugar, me gustaría comentar la conclusiones técnicas que se han extraído. Tras terminar el proyecto, se puede afirmar que se ha cumplido el objetivo principal que se ha planteado en la sección 1.2. Mediante un proceso de investigación, se ha podido diseñar e implementar un modelo de máquinas de estados finitos capaz de controlar un agente automático para el TORCS. A través de esta modelo se controlan algunos de los sensores del coche y se consigue gestionar el mismo para que sea capaz de circular por la pista e, incluso, competir contra algunos de los rivales. Al tratarse de un único objetivo principal, la parte de trabajo recae en el proceso de investigación y diseño de la MEF para encontrar el mejor modelo posible; mientras que la parte de implementación, aunque es costosa por la cantidad de código que hay que entender y generar, resulta algo más trivial.

Me gustaría hacer cierto énfasis en el proceso de investigación sobre qué lenguaje de programación utilizar. Aunque se ha tratado de un periodo breve, han surgido ciertas complicaciones en con los lenguajes de Matlab y Python, y por ello se terminaron desestimando y se abordó el problema con Java.

Tras observar los resultado obtenidos en la sección de pruebas (6), se puede concluir que la solución obtenida era la esperada frente al objetivo, pudiendo existir posibles mejoras que se puedan abordar en trabajos futuros.

En cuanto a las conclusiones personales, me gustaría destacar que acabo este capítulo de mi vida con cierto grado de satisfacción, pero teniendo en cuenta que podría haber obtenido unos resultados personales mejores. Con respecto a la carrera, tras 5 años de trabajos, sacrificios y descansos, me he dado cuenta de que siempre se puede dar un grado más de uno mismo en las tareas. Creo que podría haber culminado

mi carrera académica con una mayor grado de conocimiento, pero espero que con el nuevo capítulo que abriré tras este proyecto me aportará las aptitudes que me faltan. Centrándome en el proyecto que se describe en este documento, me gustaría destacar que ha supuesto un reto personal. Hace casi 2 años estaba luchando por realizar un proyecto parecido para una de las asignaturas de la carrera, de la cual podría decir que fue de las que más me hizo sufrir. Al tomar este proyecto, he recordado algunos de esos momentos. Pero, al terminarlo, me siento satisfecho conmigo mismo y puedo zanjar esta etapa con una sonrisa.

8.2. Trabajos futuros

El desarrollo de este agente automático supone un aporte para la comunidad el cuál puede ser utilizado en diversos campos. Por una parte, se puede continuar con el desarrollo del mismo e intentar perfeccionar algunos de sus aspectos, mejorar las MEF que lo controlen o combinarlo con otras técnicas de la inteligencia artificial como las redes de neuronas o las máquinas de Turing.

El proyecto en general puede ser utilizado para la implementación de agentes automáticos en nuevos videojuegos, o, al menos, servir como una base para estos. Cambiando de campo, este trabajo otorga unos resultados que sirven como análisis y pruebas previas al desarrollo de coches autónomos.

En un enfoque académico, este proyecto puede ser utilizado tanto para la enseñanza del funcionamiento de las máquinas de estados, así como en el desarrollo de videojuegos.

Bibliografía

[1] E. Jurado Málaga, *Teoría de Autómatas y Lenguajes Formales*. Cáceres, España: Universidad de Extremadura, UEX, 2008.
URL: http://158.49.113.108/bitstream/handle/10662/2367/978-84-691-6345-0.pdf?sequence=1&isAllowed=y.

[2] S. Russel and P. Norvig, *Aritficial Intelligence: A modern aproach*. New Jersey: Prentice-Hall, 1995.
URL: https://pdfs.semanticscholar.org/bef0/731f247a1d01c9e0ff52f2412007c143899d.pdf.

[3] GNU GPL, URL: https://www.gnu.org.

[4] D. A. Reyes Gómez, "Descripción y aplicaciones de los autómatas celulares," tech. rep., FES Acatlán, UNAM, agosto 2011.

[5] D. F. Gillies, "Traffic lights: A desing example," tech. rep., Imperial College London, 2015.

[6] A. Monga and B. Singh, "Finite state machine based vending machine controller with auto-billing features," *International Journal of VLSI design & Communication Systems (VLSICS)*, vol. 3, abril 2012.

[7] A. Alrehily, R. Fallatah, and V. Thayananthan, "Design of vending machine using finite state machine and visual automata simulator," *International Journal of Computer Applications*, vol. 115, abril 2015.

[8] Control Programable de Lavadora, URL: http://cc.etsii.ull.es/ftp/antiguo/IDL-gestion/Proyecto_sensorinteligente/sed/CONTROL%20PROGRAMABLE%20DE%20LAVADORA_05-06.pdf.

[9] Siri, Apple, URL: https://www.apple.com/es/ios/siri.

[10] Apple, URL: https://www.apple.com.

[11] Google Home, Google, URL: https://madeby.google.com/intl/en_us/home.

[12] A. Griffin, "Facebook's artificial intelligence robots shut down after they start talking to each othe in their own language," *Independent*, julio 2017.

[13] ESA, "Essential facts about the computer and video game industry," 2017. URL: http://www.theesa.com/wp-content/themes/esa/assets/EF2017_Design_FinalDigital.pdf.

[14] NewZoo, "2016 global games market report: an overview of trends & insights," junio 2016. URL: https://cdn2.hubspot.net/hubfs/700740/Reports/Newzoo_Free_2016_Global_Games_Market_Report.pdf.

[15] S. Belli and C. López Raventós, "Breve historia de los videojuegos," *Athenea Digital, Revista de Pensamiento e Investigación social*, vol. 14, pp. 159–179, otoño 2008. URL: http://www.redalyc.org/pdf/537/53701409.pdf.

[16] J. Alcalá, "Inteligencia artificial en videojuegos." Ciclo de conferencias Game Spirit 2, URL: http://www.flasentertainment.com/blog/ia.pdf.

[17] Unity, URL: https://unity3d.com/es.

[18] Unreal Engine, URL: https://www.unrealengine.com/en-US/what-is-unreal-engine-4.

[19] Epic Games, URL: https://www.epicgames.com.

[20] NaughtyDog, "State-based scripting in uncharted 2: Among thieves." *Presentación*, URL: https://es.slideshare.net/naughty_dog/statebased-scripting-in-uncharted-2-among-thieves.

[21] R. N. Saucedo Delgado. GitHub, URL: https://github.com/norman-ipn.

[22] M. Rimachi, L. Bardalez, and D. Achanccaray, *Desarrollo de un Videojuego en Tercera Persona utilizando Máquinas de Estado Finitas (FSM) y Pathfinding Jerárquico (HPA)*. PhD thesis, Universidad Nacional de Ingeniería, diciembre 2011. URL: http://www.buenastareas.com/ensayos/Desarrollo-De-Un-Videojuego-En-Tercera/3277781.html.

[23] V. Giannaccini Sapsezian. *Pseudo-Portfolio*, URL: https://munchyfly.me.

[24] V. M. Zamora España, "Gestión de rutas y toma de decisiones en el entorno de simulación sti-sim." TFG, Universidad Carlos III de Madrid (UC3M), junio 2016.

[25] B. Wymann, E. Espié, C. Guionneau, C. Dimitrakakis, R. Coulom, and A. Sumner, "TORCS, The Open Racing Car Simulator," 2014. URL: http://www.torcs.org.

[26] B. Wymann. Site: http://www.berniw.org.

[27] SourceForge TORCS, URL: https://sourceforge.net/projects/torcs.

[28] M. Reza Bonyadi, S. Nallaperuma, D. Loiacono, and F. Neumann. SCRC, (S)imulated (C)ar (R)acing (C)hampionship, URL: http://cs.adelaide.edu.au/~optlog/SCR2015.

[29] Universidad de Adelaida, URL: http://cs.adelaide.edu.au.

[30] Universidad Politécnica de Milán, URL: https://www.polimi.it/home.

[31] Proyecto CIG, URL: https://sourceforge.net/projects/cig.

Anexo

Anexo I: Manual de usuario

A continuación se presentará el manual de usuario para la descarga de los archivos, la instalación del juego y el uso del sistema.

Descarga de archivos

En primer lugar, se puede acceder a la página principal del videojuego (TORCS[25]) para comprobar las últimas versiones disponibles así como diferente información del juego (enlaces de descarga, FAQ, multimedia, etc.). También se puede acceder a esta información a través de la página de *SourceForge* del proyecto del TORCS donde se encuentran todos los archivos de descarga referentes al videojuego y diversa información como una *Wiki* (bastante escasa), noticias sobre actualizaciones o un foro de debate y ayuda[27]. Para la parte del servidor, es necesario acceder a uno de los repositorio de competición. Nosotros utilizaremos el que viene en el proyecto de *Computational Intelligence in Games* (CIG)[31]. Por otro lado, encontramos una de las páginas de competición (SCRC[28], es la que yo he utilizado, pero hay otras más antiguas) en la que podemos obtener también algunas reglas de competición y diferentes clientes que hay en distintos lenguajes de programación (Java, C++, Python, etc.).

Instalación del cliente

Como se ha explicado en el capítulo 4 (Arquitectura y diseño del sistema), la arquitectura del videojuego se compone de cliente y servidor. En este apartado del manual se explica cómo instalar la parte del servidor.

Existen 2 versiones del videojuego dependiendo del sistema operativo: una para Linux y otra para Windows. La versión que he utilizado es la de Windows y es para la cual se especifica este manual. Si se quiere saber más a cerca de la versión de linux se puede hacer a través de la página oficial o consultando el manual que se encuentra en la página del *SCRC*.

Para descargar e instalar el juego y el servidor, existen 2 formas diferentes, una que viene todo incluido y otra en la que no. En este proyecto se ha utilizado la segunda forma ya que para la primera se necesita tener software adicional. Si se quiere conocer cómo instalar a través del primer método se puede consultar en el apartado de "Descarga/Instalación" (*Download/Instalation*) de la página oficial.

Descargas

1. **Versión del juego:** desde la página de *SourceForge* del TORCS, en el apartado de ficheros acceder al fichero `torcs-win32-bin` donde encontraremos las diferentes versiones disponibles del videojuego. Como se ha indicado en el Entorno operacional, en este proyecto se ha utilizado la versión 1.3.4, pero se puede utilizar cualquier otra que esté disponible. Hay que tener en cuenta que para archivos opcionales que se quieran descargar, es recomendable obtener los que sean de la misma versión.

2. **Parche del servidor:** se puede descargar desde la página de *SourceForge* de CIG. En el apartado de ficheros acceder a *SCR Championship/Server Windows* donde encontraremos las diferentes versiones disponibles del servidor. En este proyecto se ha utilizado la versión 1.0. Existen otras versiones de competiciones más antiguas, así como ficheros que se han subido en ese proyecto, ya que no se trata del directorio oficial del TORCS.

3. **Cliente:** para esta parte existen diferentes opciones y se puede escoger la que más agrade según el objetivo que se tenga. El cliente puede descargarse o crearse por uno mismo (esta última opción requiere entender cómo funciona el servidor, por lo que si se quiere evitar esto o interesa centrarse sólo en la parte de programar un cliente, se recomienda descargar alguno y adaptarlo al gusto). En este proyecto se ha descargado el cliente en Java desde CIG.

Instalación

1. **Instalar el juego:** comenzar la instalación del TORCS a partir del primer archivo descargado (Descargas 1).

2. **Descomprimir el parche del servidor:** mover el segundo archivo descargado (Descargas 2) a la carpeta principal del TORCS y descomprimirlo. Durante el proceso se preguntará si desea sobrescribir algunos archivos; decir sí a todo. En este punto se puede comprobar si se ha instalado todo correctamente de la siguiente forma:

   a) Ejecutar el archivo `wtorcs.exe` desde el directorio principal o ejecutar el juego desde el enlace directo creado durante la instalación (si existe).

   b) En la pantalla principal del juego seguir la siguiente ruta:

   $$\text{Race} \rightarrow \text{Quick Race} \rightarrow \text{Configure Race} \rightarrow \text{Accept}$$

   c) Si la instalación y la aplicación del parche se han realizado con éxito, se podrá observar en la lista de "Jugadores no seleccionados" (*"Not Selected Players"*) 10 instancias de `scr_server`. Es posible que una de ellas (la primera) ya se encuentre seleccionada en la lista de "Jugadores seleccionados" (*"Selected Players"*).

3. **Compilación del cliente:** como ya se ha dicho, existen diferentes clientes (tantos como se quieran/puedan programar). En este caso se indicarán los pasos para la compilación del cliente en Java.

a) Descomprimir el archivo del cliente que se ha descargado (se puede descomprimir fuera del directorio principal del juego). Se obtendrá la carpeta `scr-client-java`.

b) Dentro del nuevo directorio, acceder a la carpeta que contiene el código (`src`).

c) Abrir la consola de comandos en este directorio y compilar los archivos mediante el siguiente comando:

```
$ javac -d ../classes scr/*.java
```

Con esto se compilarán todos los archivos `.java` en la carpeta `classes`.

Uso del sistema

1. **Ejecución**

   I. **Ejecutar el juego:** es necesario lanzar el juego antes que el cliente para que el servidor esté activo y así el cliente no se tenga que quedar esperando. Para ejecutar el juego es necesario iniciar el fichero `wtorcs.exe` que está dentro de la carpeta del juego. Una vez iniciado, desplazarse por los menús hasta llegar al formato de carrera elegido.

   II. **Ejecutar el cliente:** es necesario escribir el siguiente comando en la consola:

```
$ java scr.Client scr.SimpleDriver
  host:<ip>port:<p>id:<client-id>
maxEpisodes:<me>maxSteps:<ms>verbose:<v>track:<trackname>stage:<s>
```

   Más abajo (tabla 8.1: Parámetros de compilación en Java) se especifica el significado de cada parámetro.
   Una vez ejecutado, el cliente se quedará esperando a que empiece la carrera.

   III. **Iniciar carrera:** iniciar una carrera desde el menú del juego. En este momento el servidor (si está debidamente configurado) esperará alguna instancia externa de algún agente y el cliente enviará la información necesaria para conectarse al servidor.

2. **Modificación del cliente**

   I. **Modificar cliente:** existen diferentes sensores y parámetros del juego, que se explican en las tablas 8.2 y 8.3, los cuales se utilizan para modificar el comportamiento del cliente.

   II. **Modificar MEF:** para modificar la máquina de estados es recomendable guiarse con el capítulo 4 de este documento. Se aconseja diseñar un modelo de máquina de estados y a continuación implementarlo en código siguiendo las especificaciones que se indican en el capítulo 5.

Tabla 8.1: Parámetros de compilación en Java

scr.Client	controlador principal del cliente (*main* y comunicación con el servidor)
scr.SimpleDriver	controlador secundario, maneja los sensores del cliente (puede modificarse o reemplazarse)
ip	dirección IP de la máquina donde se ejecutará el servidor de competición; por defecto: *localhost*
p	puerto desde el que escucha el servidor; valores típicos: 3001-3010; por defecto: 3001
client-id	ID del agente automático; por defecto: *SCR*
me	número máximo de episodios de aprendizaje; por defecto: 1
ms	número máximo de pasos por cada episodio; por defecto: 0 (0 se utiliza para un número ilimitado de pasos)
v	nivel de verbosidad; valores: on/off; por defecto: off
trackname	nombre del circuito; por defecto: *unknown*
s	estado de la competición; valores: 0 *Warm-up*, 1 *Qualifying*, 2 *Race*, 3 *Unknown*; por defecto: *Unknown*

Anexo II: Sensores y actores del sistema

Tabla 8.2: Sensores

Sensor	Descripción
speedX	Velocidad a la que se mueve el vehículo en el eje X.
angle	Ángulo de orientación del vehículo frente a los ejes del trazado.
track	Valor de ”visión” de cada ángulo del coche (-90 a 90). Muestra la distancia hasta el objeto más cercano.
focus	-
trackPos	Posición dentro del trazado.
gear	Marcha engranada.
rpm	Revoluciones por minuto del motor.
opponents	Distancia a los oponentes.
racePos	Posición dentro de la carrera.
speedY	Velocidad del vehículo en el eje Y. Velocidad lateral.
curLapTime	Tiempo transcurrido de la vuelta actual.
damage	Nivel de daño que ha recibido el vehículo.
distFromStart	Distancia hasta la línea de llegada.
distRaced	Distancia recorrida desde el inicio de la carrera.
fuel	Nivel de combustible.
lastLapTime	Tiempo realizado en la última vuelta.
wheelSpinVel	Velocidad de giro de las 4 ruedas. Muestra 4 valores diferentes, 1 para cada rueda.
speedZ	-
z	-

Los sensores con color azul son los principales y más a tener en cuenta a la hora de desarrollar un agente que sea capaz de circular por la pista. Los sensores de color naranja aportan una información adicional que puede llegar a no ser relevante para la lógica del agente. Los sensores en color rojo especifican información técnica del vehículo que no es necesaria para tener un control sobre el mismo, aunque sí podrían ser utilizados en agentes muy sofisticados.

Tabla 8.3: Actores

Actores	Descripción
accel	Nivel de aceleración del vehículo. Valores: de 0 a 1
brake	Nivel de freno del vehículo. Valores: de 0 a 1
clutch	Nivel de embrague del vehículo. Valores: de 0 a 1
gear	Marcha del vehículo. Valores: de -1 (marchar atrás) a 6
steer	Dirección del vehículo. Valores: de -1 a 1
focus	-

Los valores que contienen un guión (-) en su descripción son desconocidos por la falta de información del juego o debido a que no tienen la relevancia necesaria. Exceptuando las marchas, el resto de valores son de tipo flotante.